One-Phase Real-Time Commit Processing

A Dissertation
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in
Computer Science & Engineering

By
Prabodh Saha

Department of Computer Science and Automation
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

July 1999

To

My Parents

Abstract

To ensure *transaction atomicity*, Distributed Database Systems implement *transaction commit protocols*. A commit protocol guarantees the uniform commitment of distributed transaction execution, that is, it ensures that all the participating sites agree on the final transaction outcome (commit or abort). Most importantly, this guarantee must be valid even in case of site or network failures. Database researchers have been working in this area for the last three decades and a variety of commit protocols have so far been proposed. These protocols include one-phase protocols like *Early Prepare(EP)*, two-phase protocols like the classical *Two Phase Commit(2PC)*, three-phase protocol *Three Phase Commit(3PC)* and many of their optimizations.

These protocols require exchange of multiple messages, in one or more phases, between the participating sites where the distributed transaction is executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to the disk immediately. Due to these costs, commit processing can result in a significant increase in transaction execution time. The commit protocols hold data-locks for a substantial fraction of its total commit processing time. This leads to priority inversion, which affects the real-time performance of the system considerably. Consequently, the choice of commit protocol is a specially important design decision for distributed Real-Time DataBase Systems (RTDBS).

We profiled the real-time performance of the Early Prepare commit protocol and compared its performance with the above-mentioned 2PC protocol and very recent optimistic real-time commit protocol PROMPT, using a detailed simulation model of a distributed RTDBS.

Our study was carried out in the context of real-time applications with *firm deadlines*, wherein transactions that miss their deadlines are considered to be worthless and are immediately "killed", that is, aborted and discarded from the system without being executed to completion. Accordingly, the performance metric is *KillPercent*, the steady-state percentage of killed transactions.

Our experimental results reaffirmed that distributed commit processing can have considerably more effect than distributed data processing on the real-time performance, thereby highlighting the need for developing commit protocols tuned to the real-time domain. Moreover the results show an interesting "split personality" behavior: The one-phase protocols perform very well for parallel distributed transactions, outperforming PROMPT,the best-performing two-phase real-time commit protocol. For sequential distributed transactions, however, they perform rather poorly.

Acknowledgements

I would like to take this opportunity to express my heartfelt gratitude to my guide Dr. Jayant Haritsa. The vastness of his knowledge, attitude towards perfection and the zeal with which he keeps abreast with the latest developments in the diverse research areas relating Databases, is simply admirable.

The Database Systems Lab itself has been a constant source of inspiration for me. Caring lab-mates and extremely well maintained facilities make the place an ideal environment to work in.

Words fail to express how indebted I am to my *parents* and my *sister*. I wish I knew some way of returning even a fraction of what they have given me.

Contents

List of Figures

List of Tables

CHAPTER

1

Introduction

A *transaction* is the fundamental unit of data accessing and modification in a database management system. Every transaction is an atomic unit of processing. The successful execution of a transaction results in the transaction being *committed*. This means all its effects on the database are permanent regardless of possible subsequent system failures. In case a transaction cannot commit, for whatever reason, all the effects created by the partial execution of the transaction are removed from the database and the transaction is said to have been *aborted*. Thus, "either all or none" of the effects of the transaction will persist in the system.

For some practical reasons, many database applications store their data *distributed* over a number of sites that are connected by a communication network. In such a set up, a single transaction may have to execute in many sites, depending on the location of the data that it needs to process. In such cases, some sites of a transaction may decide to commit while the other could decide to abort it, resulting in a violation of transaction atomicity. To address this problem, distributed database systems use transaction commit protocols. The job of the commit protocol is to make sure that all the participating sites of a transaction agree upon the final outcome (commit or abort) of the transaction. Most importantly, this assurance has to remain valid in the presence of site or network failures.

Over the last two decades, a variety of distributed commit protocols have been proposed by database researchers including the one-phase *Early Prepare(EP)* [SC90, SC93], the classical *Two Phase Commit(2PC)* [LS76, Gra78] and many of its variations, also, the *three phase commit* protocol to get rid of blocking in case of network or site failures.

All the commit protocols require exchange of multiple messages, in one or more phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to the disk immediately. Due to these costs, commit processing can result in a significant increase in transaction execution times [SBCM95, SJR91]. For a large fraction of the commit processing time, the data-locks are held by distributed transactions. This leads to priority inversion, where a higher priority transaction with closer deadline has to wait for a low priority transaction to finish, which considerably affects the real-time performance of the system. As a result, the choice of commit protocol bcomes an extremely important design decision for distributed Real-Time DataBase Systems (RTDBS).

To address the above issue, the design of high-performance real-time distributed commit protocols were investigated by [GHR97, HRG98]. They have come up with a two-phase based optimistic commit protocol "PROMPT" [HRG99] which has proved to perform better than all other two-phase protocols in the field of distributed real-time database systems. However, their study did not include one-phase protocols into consideration. They expected one-phase protocols to perform poorly in distributed RTDBS. This is what we take into account and study the performance of one-phase protocols in this thesis.

Our study is carried out in the context of real-time applications with *firm deadlines* [HCL92], wherein transactions that miss their deadlines are considered to be worthless and are immediately "killed", that is, aborted and discarded from the system without being executed to completion. Accordingly, the performance metric is *KillPercent*, the steady-state percentage of killed transactions.[1]

In our experimental results, we see that distributed commit processing can have considerably more effect than distributed data processing on the real-time performance . Moreover the results show an interesting "dual behavior" of the one-phase commit protocols. They perform very well for parallel distributed transactions, beating the best-performing two-phase protocol PROMPT. However, for sequential distributed transactions, they per-

[1] Or, equivalently, the percentage of missed deadlines.

form rather poorly.

1.1 Contributions

In this thesis, we quantitatively investigate the performance implications of supporting transaction atomicity in distributed RTDB systems by performing an extensive set of simulation experiments. Our contributions are two fold:

- Using a detailed model of a distributed firm-deadline real-time database system, we profile the real-time performance of a representative suite of commit protocols studied in [Gup97] and extend the study for the one-phase commit protocols. *To the best of our knowledge, these are the first quantitative results for the one-phase protocols in this area.*

- We reveal an interesting "split personality" behavior of the one-phase protocols. The one-phase protocols perform very well for parallel distributed transactions, outperforming the best-known two-phase real-time commit protocol. For sequential distributed transactions, however, they perform rather poorly. There the optimistic protocol PROMPT wins over EP for wide range of workload that covers any workload that can normally be expected in real life.

1.2 Organization

The remainder of this thesis is organized as follows. An overview of Distributed Database System is given in the Chapter 2. Then we give an overview of Real Time Database Systems (RTDB) in Chapter 3. The PROMPT and EP protocols along with the classical 2PC protocol are described in Chapter 4. The simulation model is described in Chapter 5 and the results of the simulation experiments, which chiefly compare PROMPT and EP, are highlighted in Chapter 6. Finally, in Chapter 7, we present the conclusions of our study.

CHAPTER

2

Distributed Database Systems

Distributed Database Systems technology is the intersection of two technologies, namely, *database systems* and *computer networks*

2.1 What is a Distributed Computing System ?

Distributed Computing System consists of a number of processing elements (not necessarily homogeneous) that are interconnected by a computer network and that co-operate in performing their assigned tasks.

Distributed Computing Systems can be classified with respect to a number of criteria, like: *degree of coupling*, *interconnection structure*, *interdependence of components* and *synchronization between the components.*

2.2 What is a Distributed Database System ?

A distributed database can be defined as a collection of multiple, logically interrelated databases distributed over a computer network. A Distributed Database Management System is a *Software System* that permits the management of the Distributed Database System and makes the distribution transparent to the users.

A typical Distributed Database will have the following features:

- Entire database is distributed over a number of distant sites, potentially having replication of some portion of data .
- In every site there exists means to access data situated in a remote site.
- The records kept in a location is more frequently accessed by the locally submitted transactions than by remote transactions.

2.3 Advantages of Distributed Databases

The Distributed Databases provide a number of advantages, like

1. Local autonomy
2. Improved Performance (by proper fragmentation)
3. Improved reliability/availability (via replication)
4. Greater expandability
5. Greater shareability

2.4 Disadvantages of Distributed Databases

The distributed nature of the database systems provide many problems, like

1. Higher complexity
2. Consequently higher hardware and software cost
3. Synchronization and co-ordination problem amongst the sites
4. Higher maintenance overhead in case of replication
5. Greater security problems

CHAPTER

3

Overview of RTDB Systems

For the past decade, there has been a growing interest in the area of *Real-Time Database Systems (RTDB)*. The importance of research in this area has been emphasized in many workshops and conferences at the international fora in recent times [Son88, Son90, AGM92, Ulu92, Ulu94b, Bes96, BH95, BH96]. An RTDB system is an amalgam of a conventional real-time system and a database system. In a real-time system, the tasks have associated timing constraints, usually in the form of *completion deadlines*, and are scheduled in a way that they can be completed before their deadlines expire. Preserving database consistency is usually not considered in conventional real-time systems. On the other hand, timing constraints of the transactions are usually not addressed in conventional database systems. In an RTDB system, the transaction processing must not violate the database consistency, like conventional database systems. In addition, the RTDB system has to meet the timing requirements of the transactions. In other words, in order to successfully commit a transaction, the RTDB system has to satisfy the timing constraints imposed on the transaction commitment in addition to the logical consistency requirements of the transaction. The goal in an RTDB system usually is to maximize the *number* of transactions that complete before their deadlines expire, as opposed to minimizing the average transaction response time (or maximizing the average transaction throughput) which is the primary goal in a conventional database system.

Material for this chapter is taken from [Gup97].

3.1 Real-Time Framework

The applications in the real-time domain can be grouped into the following three categories based on how the application is impacted by the violation of the task completion deadline [HCL92]:

Hard Deadline Real-Time Applications: In these applications, the consequences of missing the deadline of even a single task could be catastrophic. Life-critical applications such as flight control systems or missile guidance systems belong to this category. Database systems for efficiently supporting hard deadline real-time applications, where *all* transaction deadlines have to be met, appear infeasible due to the large variance between the average case and the worst case execution times of a typical database transaction. The large variance is due to transactions interacting with the operating system, the I/O subsystem, and with each other in unpredictable ways. Guaranteeing completion of all transactions within their deadlines under such circumstances requires an enormous excess of resource capacity to account for the worst possible combination of concurrently executing transactions.

Soft Deadline Real-Time Applications: In these applications, the tasks do have associated deadlines, but even if a task fails to complete within the deadline, it is allowed to execute upto completion. Generally, in these systems, a "value function" assigns a value to the tasks. This value remains constant upto the deadline, but starts decreasing after the deadline. The questions to be addressed in these applications include how to identify the proper value function, which actually may be application dependent. The emphasis is on the number of tasks completing within their deadlines, as well as on the amount of *tardiness* of the tasks missing their deadlines.

Firm Deadline Real-Time Applications: These applications are different from the soft deadline applications in the sense that the tasks which miss the deadline are considered worthless (and may even be harmful if executed to completion) and are thrown out of the system immediately. The emphasis, thus, is on the *number* of tasks that complete within their deadlines.

Our interest in the RTDB systems is on the applications in the *firm deadline* real-time domain [HCL92]. We believe that understanding firm deadline RTDB systems will provide necessary insight into the RTDB technology which is necessary for addressing the more complex framework of soft deadline applications. Therefore, we have carried out our work from the perspective of a "Firm Deadline Real-Time Database System". Hereafter we will use "RTDB" to denote a firm deadline RTDB system, unless specified otherwise.

3.2 Distributed RTDB Systems

RTDB systems find applications in many areas like aerospace and military systems, computer integrated manufacturing, robotics, nuclear power plants, traffic control systems, stock market, telephone-switching systems, and network management [Son90, Ulu94a]. Many of these applications, especially in the areas of stock market, communication systems and military systems, are inherently *distributed* in nature. Unfortunately, till now, the major focus of the research in RTDB technology has been on *centralized* systems. Incorporating distributed data into the real-time database framework incurs complexities associated with concurrency control and database recovery in conventional database systems [BHG87, ÖV91]. While the issue of distributed real-time concurrency control has been considered to some extent (e.g. [SRL88, SK92, UB92]), comparatively little work has been done with regard to distributed real-time database recovery. In [Gup97] we find a detailed performance study of the implication of supporting transaction atomicity in a distributed RTDB system. However, that too is restricted to two-phase and three-phase protocols only. We have tried to fill this lacuna, considering also the one-phase protocols.

Our experimental results confirm that data distribution has a significant influence on the real-time performance, and that the choice of commit protocol clearly affects the magnitude of this influence. Among the protocols evaluated, the one-phase *early prepare* commit protocol performs markedly better than PROMPT in case of parallel execution.

However, chiefly because of its longer-data-locking feature, it suffers in the sequential execution case and performs rather poorly.

CHAPTER

4

The Commit Protocols

Distributed database systems implement a transaction commit protocol to ensure transaction atomicity. To achieve their functionality, these commit protocols typically require exchange of multiple messages and generate several log records some of which have to be force-written to the disk. Due to these costs, commit processing can result in a significant increase in transaction execution times. Moreover, commit protocols hold data-locks for a considerable part of its total commit processing time. This phenomenon results in priority inversions affecting the real-time performance and thus makes the choice of commit protocol an important design decision for the distributed RTDBS.

The conventional transaction commit protocols used in distributed Transaction Processing systems cannot be directly used in the real-time environment, because they fail to take into consideration the real-time nature of the transactions. Therefore, the conventional commit protocols need to be modified to cater to the specific requirements of the real-time transactions. In this chapter, we discuss the issues involved in adapting the conventional commit protocols to the real-time environment. We also describe a real-time optimistic commit protocol, called PROMPT[HRG99]. Then we describe, in detail, the one-phase **Early Prepare** commit protocol. To the best of our knowledge, ours is the first quantitative analysis of any one-phase commit protocol in the field of distributed RTDBS. Then we make a performance appraisal of EP and PROMPT. PROMPT was the obvious choice for comparison, because, it is known to be the best performing two-phase based commit protocol in distributed RTDBS. **Material for a part of this chapter is**

taken from [Gup97].

4.1 Distributed Transaction Execution

A common model of distributed transaction execution is shown in Figure 4.1. At the site where the transaction is submitted, a *master* process is created to coordinate the execution of the transaction. At a site where the transaction needs to access data, a *cohort* process on behalf of the transaction is created. Usually, there is only one cohort on behalf of the transaction at each such site. The master sends the STARTWORK message to a cohort when some data at the site of the cohort is required to be accessed.[1] Depending on the transaction architecture, the master may send the STARTWORK messages to multiple cohorts without waiting for their responses, that is, the cohorts of the transaction will execute in a *parallel* fashion. Or the master may send the subsequent STARTWORK message only after the previous cohort has completed the work assigned to it, that is, the cohorts of the transaction execute one after the other in a *sequential* fashion. A cohort, after successfully executing the master's request, sends a WORKDONE message to the master. After receiving the WORKDONE message, the master may send more work to the same cohort, or may decide to send the next piece of work to another cohort of the transaction.

The master may decide to conclude the transaction when all the work assigned to the transaction is completed. Or the decision may be necessitated at the instance of the user submitting the transaction. In any case, the decision to commit or abort the transaction must be uniform—the master and all cohorts must agree on a common decision. It might seem that the fact that a cohort had sent a WORKDONE message means that the cohort was willing to commit the transaction. This is true for one-phase protocols, but not for the others. The problem here is: The sending of the WORKDONE message does not always enforce any binding on the cohort to agree on the decision. Due to reasons such

[1] Actually, the master sends the STARTWORK message to the transaction manager of the site. The transaction manager creates a cohort on behalf of the master, if one is not already existing. As these implementation details are not important for the understanding of the concepts presented here, we will ignore them for the sake of simplicity.

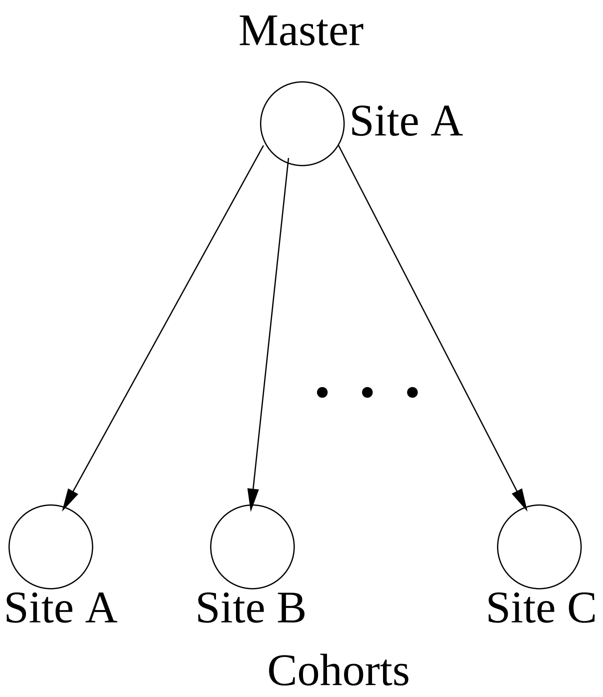

Figure 4.1: Distributed Transaction Execution

as concurrency control, performance, etc. the cohort could still be aborted even after the WORKDONE message was sent unless the protocol is a one-phase commit protocol.

4.2 The Two-Phase Commit Protocol

A distributed transaction consists of one process, called the *master*, which is executed at the site where the transaction is submitted, and a set of other processes, called *cohorts*, which execute on behalf of the transaction at the various sites that are accessed by the transaction. The master initiates the cohorts by sending STARTWORK message to them. After the cohort has finished the execution of the work assigned to it by the master, it sends a WORKDONE message to the master. All the cohorts of the transaction could be initiated together, resulting in a *parallel* execution of the transaction, or the master could send the STARTWORK messages to the cohorts one by one only after receiving the WORKDONE message for the currently executing cohort. The master initiates the *commit* protocol after receiving the WORKDONE messages from all of the cohorts.

For this model, a variety of transaction commit protocols have been devised, most of which are based on the classical **two-phase commit (2PC)** protocol.

Description of the Protocol

The master initiates the first phase of the protocol by sending PREPARE (to commit) messages in parallel to all the cohorts. Each cohort that is ready to commit first force-writes a PREPARE log record to its local stable storage and then sends a YES vote to the master. At this stage, the cohort has entered a prepared state wherein it *cannot unilaterally* commit or abort the transaction but has to wait for the final decision from the master. On the other hand, each cohort that decides to abort, writes (and not force-writes) an ABORT log record and sends a NO vote to the master. Since a NO vote acts like a veto, the cohort is permitted to unilaterally abort the transaction without waiting for a response from the master.

After the master receives the votes from all the cohorts, it initiates the second phase of the protocol. If all the votes are YES, it moves to a committing state by force-writing a COMMIT log record and sending COMMIT messages to all the cohorts. Each cohort after receiving a COMMIT message moves to the committing state, force-writes a COMMIT log record, and sends an ACK message to the master.

If the master receives even one NO vote, it moves to the aborting state by force-writing an ABORT log record and sends ABORT messages to those cohorts that are in the prepared state. These cohorts, after receiving the ABORT message, move to the aborting state, force-write an ABORT log record, and send an ACK message to the master.

Finally, the master, after receiving acknowledgments from all the prepared cohorts, writes an END log record and then "forgets" the transaction.

4.3 Firm Deadline Distributed Transactions

As mentioned earlier, in a firm deadline real-time application framework, the transactions that miss their deadlines are considered to be worthless (and may even be harmful if executed to completion) and are immediately "killed" and discarded from the system without being executed to completion. Therefore, in an ideal distributed RTDB system, the master and all the cohorts of a transaction should agree to commit or abort before the transaction deadline expires. However, in practical distributed RTDB systems, it is impossible to provide such guarantees because of the possibility of communication link or site failures and arbitrary message delays [DLW89]. In addition, there are inherent complexities involved in achieving a global time-clock for the distributed system—synchronization of the clocks [Lam78] at various sites usually is a difficult task. Therefore, to avoid the inconsistencies in such cases, in a firm deadline RTDB system, a transaction is said to be **committed** if the master has reached the commit decision (that is, force-written the `COMMIT` log record to the disk) before the expiry of the deadline irrespective of whether the cohorts have received the decision by the deadline, that is, global time is viewed from the master's perspective. Note that once the master has reached the commit decision, this decision eventually will be conveyed to the cohorts even if the deadline has expired in the process. In other words, the master must reach the decision before the expiry of the deadline. If the master is unable to reach a commit decision before the deadline expires, it makes an abort decision, i.e., it immediately aborts the transaction and notifies the cohorts about the decision. Due to arbitrary message delays, a cohort may *not* receive the final decision before the deadline expires, even though the master had dispatched it before the expiry of the deadline. If a cohort eventually receives the final decision after the expiry of the deadline, all that happens is that access to data in prepared state is prevented even beyond the deadline until the decision is received by the cohort, and other transactions which would normally expect the data to be released by the deadline only experience a delay. Typically the master is responsible for returning the results of a transaction to the invoker of the transaction. From this discussion, it is clear that such results will have been generated before the deadline if a transaction commits.

4.4 Priority Inversion

The commit protocol 2PC, described in section 4.2, was designed for conventional database systems where transaction throughput or response time is the primary performance metric.

In real-time database systems, however, satisfaction of transaction timing constraints is the primary goal. Therefore, the scheduling policies at the various resources (both physical and logical) in the system can be reasonably expected to be priority-driven with the priority assignment scheme being tuned to minimize the number of missed deadlines. The conventional commit protocols, however, do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature [SRL87]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable.

Priority inversion is usually prevented by resolving all conflicts in favor of transactions with higher priority. At the CPU, for example, a scheduling policy such as Priority Preemptive Resume ensures the absence of priority inversion. For a non-preemptable resource such as the disk, although it is not possible to completely eliminate priority inversion due to physical device restrictions, at least it is possible to *bound* the inversion period by ensuring that the wait queue for the device is processed in priority order. Even at the network, it is easy to ensure that messages of high priority transactions are transmitted before those of other lower priority requesters. With respect to data access also, priority-based pre-emptive concurrency control algorithms such as 2PL-HP [AGM88] and OPT-WAIT [HCL90] have been developed.

Removing priority inversion in the commit protocol, however, is *not* fully feasible. This is because, once a cohort reaches the prepared state, it has to retain all its data locks until it receives the global decision from the master—this retention is fundamentally necessary to maintain atomicity. Therefore, if a high priority transaction requests access to a data item that is locked by a prepared cohort of lower priority, it is not possible to forcibly obtain access by preempting the low priority cohort. In this sense, the commit phase in a distributed real-time database system is *inherently* susceptible to priority inversion. More importantly, the priority inversion is *not bounded* since the time duration that a cohort is in the prepared state can be arbitrarily long (for example, due to network delays). If the inversion period is large, it may have a significant effect on performance.

4.5 Real-Time Commit Protocols

The conventional commit protocols suffer from two problems: (a) they are not deadline cognizant and hence are not capable of meeting the transaction deadline semantics as described in Section 4.3, and (b) they do not make use of the transaction priorities at the physical and logical resources, rendering them inefficient in the real-time domain. Even though the priority inversion arising due to prepared cohorts cannot be eliminated in these commit protocols, in order to be useful in the real-time environment, these commit protocols are modified to incorporate the following:

- The commit messages have the priorities of the transactions for which they are generated, so that the messages of a higher priority transaction can be given preference over the lower priority transactions at the CPU. In addition, processing of the data pages of a higher priority transaction at the CPU is given preference over the messages of lower priority transactions.
- The master ensures that a decision about the transaction is reached by the deadline. If the deadline expires before the master had dispatched the PREPARE messages to the cohorts, it *unilaterally* decides to abort the transaction and sends the ABORT messages to the cohorts. On the other hand, if the deadline expires after dispatching the PREPARE messages, but before the master had reached a decision (i.e., all votes have not yet been received), the master makes an abort decision and the abort processing of the transaction is carried out.

These modifications to the conventional commit protocols ensure that the deadline semantics of a transaction as described in Section 4.3 are met. Also, the real-time variant of the commit protocols exploit the priorities of the transactions while contending for the resources for message processing, etc. Hereafter we will not use the term "real-time variant" explicitly while referring to the real-time variants of the commit protocols. If the non-real-time (conventional) versions of the commit protocols are to be meant, we will explicitly state so.

4.6 PROMPT: The Real-Time Optimistic Commit Protocol

The main feature of **PROMPT** (Permits Reading Of Modified Prepared-data for Timeliness) commit protocol [HRG99] is that, transaction requesting data items held by other transactions in the prepared state are allowed to access this data. That is, prepared cohorts *lend* their uncommitted data to concurrently executing transactions (without of course, releasing the update locks). In this context, three situations may arise:

Lender Receives Decision First: Here, the lending cohort (i.e., the prepared cohort) receives its global decision before the borrowing cohort has completed its local execution. If the global decision is to commit, the lending cohort completes its processing in the normal fashion. If the global decision is to abort, then the lender is aborted in the normal fashion; in addition, the borrower is also aborted since it has utilized inconsistent data.

Borrower Completes Execution First: Here, the borrowing cohort completes its execution before the lending cohort has received its global decision. The borrower is now "put on the shelf", that is, it is made to wait and not allowed to send a WORKDONE message to its master. This means that the borrower is not allowed to initiate the processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until either the lender receives its global decision or its own deadline expires, whichever occurs earlier. In the former case, if the lender commits, then the borrower is "taken off the shelf" and allowed to send its WORKDONE message. However, if the lender aborts, then the borrower is also aborted immediately since it has utilized inconsistent data. In the latter case (deadline of a borrower expires while waiting on the shelf), the borrower is killed in the normal manner.

Borrower Aborts During Data Processing Before Lender Receives Decision: Here, the borrowing cohort aborts in the course of its data processing (due to either a local problem, deadline expiry or receipt of an ABORT message from its master) before the lending cohort has received its global decision. In this situation, the borrower's updates are undone and the lending is nullified.

In summary, the protocol allows transactions to access uncommitted data held by lower priority prepared transactions in the "optimistic" belief that this data will eventually be committed.

To farther improve its real-time performance, three additional features were included in the PROMPT protocol: Active Abort, Silent Kill and Healthy Lending. These features are described below.

4.6.1 Additional Features of PROMPT

The following features were also included in the PROMPT protocol since they are expected to improve PROMPT's real-time performance:

Active Abort : In the basic 2PC protocol, cohorts are *passive* in that they inform the master of their status only upon explicit request by the master, that is, if the cohort is aborted after it had dispatched the WORKDONE message, it will inform the master only on receiving the PREPARE message from the master. In a conventional distributed DBMS, after a cohort has finished its work and sent the WORKDONE message to the master, it cannot be aborted due to local data conflicts or serializability violations (assuming a lock-based concurrency control mechanism). However, in a distributed RTDB system, even such a cohort can be aborted due to higher priority conflicting transactions (provided the cohort is not already in the prepared state). Therefore, in a real-time situation, it may be better for an aborting cohort to immediately inform the master so that the abort at the other sites can be done earlier. Hence, cohorts in PROMPT inform the master as soon as they decide to abort locally.

Silent Kill : For a transaction "kill", that is, an abort that occurs due to missing the deadline (before the master dispatched PREPARE messages), there is no need for the master to invoke the abort protocol since the cohorts of the transaction can independently realize the missing of the deadline (assuming global clock synchronization). Therefore, in PROMPT, aborts due to deadline misses—before a cohort enters the prepared state—are done "silently" without requiring any communication between the master and the cohorts.

Healthy Lending : A committing transaction that is close to its deadline may be killed due to deadline expiry before its commit processing is over. Lendings by such transactions must be avoided since such aborts can result in the aborts of all the associated borrowers. To address this issue, we have added a feature to PROMPT whereby only transactions whose deadlines are not very close, that is, "healthy" transactions are allowed to lend their prepared data. This is realized in the following manner: A *health factor* H_T is associated with each transaction T and a transaction is allowed to lend its data only if its health factor is greater than a (system-specified) minimum value $MinHF$. The health factor is computed at the point of time when the master is ready to send the PREPARE messages and is defined to be the ratio *TimeLeft* / *MinTime*, where *TimeLeft* is the time left until the transaction's deadline, and *MinTime* is the minimum time required for commit processing (recall that a minimum of two messages and one force-write need to be processed before the master can take a decision).

The success of the above scheme is directly dependent on the threshold health factor $MinHF$ – set too conservatively, it will turn off the borrowing feature to a large extent, thus effectively reducing PROMPT to standard 2PC; on the other hand, set too aggressively, it will fail to stop several lenders that will eventually abort.

4.6.2 No Cascading Aborts in PROMPT

Even though access to uncommitted data is allowed, aborts in PROMPT do not cascade. The problem of cascading aborts in PROMPT is alleviated due to the following reasons:

1. The lending cohort is in prepared state and cannot be aborted due to local data conflicts (we assume a lock-based concurrency control mechanism). Therefore, if

the lending transaction finally decides to abort, it does so because of two reasons (in addition to consistency violations): (a) the master has already sent the PREPARE messages, but the deadline expires before it is able to reach a decision (Section 4.3), or (b) any of the siblings of the cohort in the prepared state votes NO. Note that when a cohort is aborted, it immediately sends an ABORT message to master (Active Abort policy), and the master, after receiving an ABORT message from a cohort never sends the PREPARE messages to other cohorts. Therefore, the situations in which a sibling of the prepared cohort could vote NO can be the following: (i) the ABORT message sent by the sibling cohort was in *transit* when master dispatched the PREPARE messages, or (ii) the PREPARE message sent by the master to the sibling cohort was in transit when the cohort was aborted. As the time during which a message is in transit is usually small compared to the transaction execution times, the above situations are unlikely to occur frequently. Hence, a lending transaction is typically expected to commit.

2. Even if the lender eventually does abort, it only results in the abort of the immediate borrower(s) and does not cascade beyond this point (since borrowers are not in the prepared state which is the only situation in which uncommitted data can be accessed).

In short, the abort chain for PROMPT is bounded and is of length one (of course, if an aborting lender has lent to multiple borrowers, then all of them will be aborted, but the abort chain for each borrower is bounded as described above).

4.7 Early Prepare Commit Protocol

The Early Prepare Commit Protocol uses Presumed Commit (PC: an optimization over 2PC, where a cohort in prepared state goes ahead to commit in case of "no reply" from its master) to eliminate one round of messages for a distributed transaction that executes in the absence of failures.

EP also reduces the communication further by making each cohort enter the prepared state after it performs its work, and before it replies to the master with the WORKDONE message.

A master using EP may have to force multiple `MEMBERSHIP` records, because the transaction membership may grow as transaction execution progresses. A master using the presumed commit protocol must record the identity of a cohort in its stable log before the cohort can enter its prepared state. Hence a Master using EP must record a Cohort's identity in its stable log before sending a work request to that cohort. If the master knows the transaction membership before it begins executing the transaction, the master can force one membership record[2]. On the other hand, if the master uses the results obtained from one cohort to determine the identity of a subsequent cohort, and no two operations are done by the same cohort, the master must force a `MEMBERSHIP` record before sending each transaction operation to the appropriate cohort.

The steps of EP's execution are as shown below:

- Master forces one or more `MEMBERSHIP` log records and sends a STARTWORK request to each cohort.

- Each cohort executes its work request, forces a `PREPARE` log record, and replies to master with a WORKDONE message.

- Master forces a `COMMIT` log record, sends a COMMIT message to each cohort, and forgets about the transaction.

- Each cohort appends to its log (but need not force) a `COMMIT` log record and then forgets about the transaction.

When EP is used and several work requests are sent to each cohort, each cohort forces several `PREPARE` log records. In contrast , when 2PC or PC is used, each cohort involved in a transaction forces only one `PREPARE` log record regardless of the number of work requests it received.

[2]This assumption has been considered and implemented in our simulation.

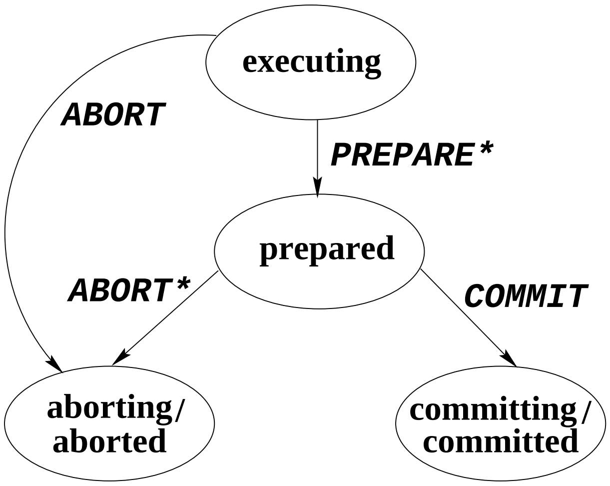

Figure 4.2: State Transitions in EP for Cohort

A cohort using EP synchronously forces each PREPARE record as early as possible, while a cohort using PC synchronously forces the PREPARE record as late as possible. A cohort using an intermediate approach could reply to the master after executing its work request, force the PREPARE record asynchronously, and notify the master once the record has been forced. This asynchronous approach reduces the number of synchronous log forces and may increase parallelism. It requires one more round of messages compared to EP, but one less round compared to PC.

4.7.1 State Transitions

Let us assume, without loss of generality, that both the master and the cohorts are in an executing state. From this point, a cohort can move to either aborting or prepared state, depending upon whether it successfully finishes its work or not. If it finishes its job successfully, it forces a PREPARE log record and sends a WORKDONE message to its master. In the Figure 4.2, the names on the transaction arcs, indicate the log records, an * with the name indicating that the log record is force-written. Once a cohort is in prepared state, it can no longer *unilaterally* decide to move to the aborting state; it has to wait for the master's decision of commit or abort, based on which it will move to aborting or committing state by force-writing the appropriate log record.

The master makes the decision (commit or abort) about the fate of the transaction.

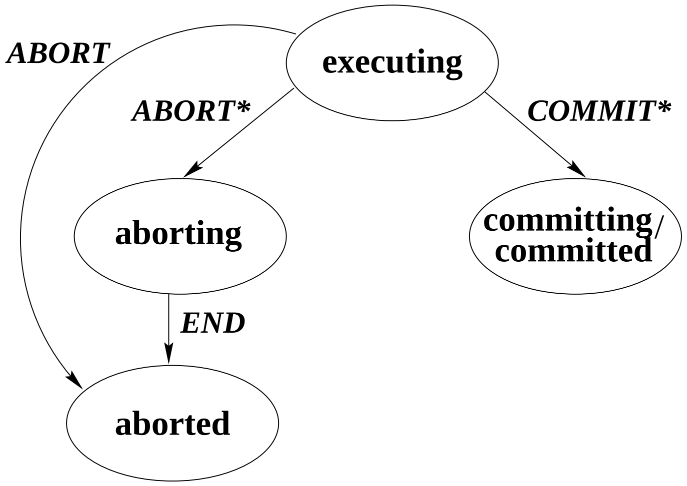

Figure 4.3: State Transitions in EP for the Master

Its state transitions are shown in Figure 4.3. It can directly move to the aborted state from the executing state sending ABORT messages to the cohorts. (for example, if the transaction is aborted by the user). Master can as well move to the aborting state by sending ABORT message to all of its successful cohorts. It moves to the committing state by sending the COMMIT messages to all the cohorts. Note that the master requires to force write the ABORT log record while moving to its "aborting" state. From the executing state, the master moves to the committing state (by writing a COMMIT log record) only if it gets all WORKDONE messages from all the cohorts and then it forgets about the transaction (because of the PC feature of EP). Even if one ABORT message is received from any cohort, the master moves to the aborting state (by force-writing an ABORT log record). From the aborting states, the master moves to the aborted state, by writing an END log record. The END log record is written only after the master has received all ACKs from the cohorts that were sent the decision. Note that, in Figure 4.2 the cohorts do not write any more log records (that is, no END record) after moving to committing or aborting states, therefore, for the cohorts, these states are equivalent to committed or aborted states, respectively.

Table 4.1: Message Overheads for EP & 2PC

	Master to Cohort		Cohort to Master	
	EP	2PC	EP	2PC
For Committing Transactions	1	2	0	2
Cohort Unsuccessful	1	1	0	1
Cohort Successful but Master Decides Abort	1	2	1	2

Table 4.2: Logging Overheads for EP & 2PC

	Master				Cohort			
	Total Log Records		Force-writes		Total Log Records		Force-writes	
	EP	2PC	EP	2PC	EP	2PC	EP	2PC
For Committing Transactions	2	2	1	1	2	2	1	2
Cohort Unsuccessful	2	2	1	1	0	1	0	0
Cohort Successful but Master Decides Abort	2	2	1	1	2	2	2	2

4.7.2 Summary of Early Prepare Commit Protocol

Like other commit protocols, EP guarantees the uniform commitment of the distributed transaction by means of messages and logging. In EP a commit decision is reached if all cohorts have performed their jobs successfully and thereby ready to commit, otherwise an abort decision is taken. A cohort may query the master to find out the outcome of the transaction. EP ensures that the master will have the information available to answer such queries because otherwise it could lead to non-uniform commitment of the transaction.

The message and the logging overheads in EP and 2PC are listed in Tables 4.1 and 4.2, respectively. Here we elaborately discuss the EP overheads and then compare its overheads with that of 2PC.

<u>EP overheads</u>

For each committing transaction the master sends 1 message to each cohort and writes two log records one of which is forced. A cohort sends no messages (as it follows PC) to the master and writes two log records one of which is forced. Note that the WORKDONE message was sent when the cohort was in its executing state. So this message has not been considered in commit processing overheads. For an aborting transaction (the transaction that aborts in the executing phase), the master sends one (abort) message to each cohort that performed its assigned work and writes two log records one of which is forced. Each unsuccessful cohort gets aborted while executing, so it sends zero message in its commit phase, and writes no log record. On the other hand, each cohort that sends WORKDONE sends one message and writes two log records, both of which are forced.

<u>Comparison with 2PC</u>

As is seen in Table 4.1 the message overhead of 2PC is markedly higher in case of 2PC. It is expected because 2PC is having an extra phase over EP. However, as is seen in the Table 4.2, the difference in the number of log records is marginal. Log records are meant to save enough information, so that the protocol does not fail in case of site or network failure. As an overall effect, the total commit processing overhead for EP is markedly less than that of 2PC.

4.8 Performance Study of Commit Protocols

As of now, very little work have been done in studying the performance of various commit protocols in distributed RTDB systems. The sole effort made in this regard [Gup97], restricts its study to the two-phase and three-phase commit protocols. In this thesis we have considered the one-phase Early Prepare(EP) commit protocol. We study its performance in an extensive set of "transaction execution environments" and for "different system configurations".

The Early Prepare commit protocol seems to be of importance, especially in real time scenario. We make a comparison between the one-phase EP and the best-performing two-phase protocol PROMPT. Following are the aspects:

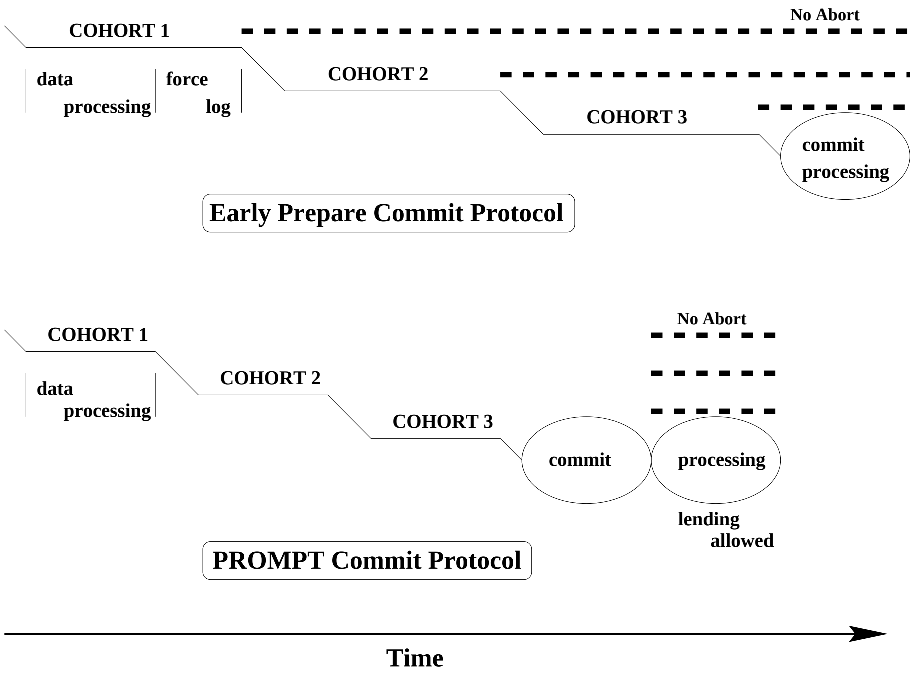

Figure 4.4: Sequential Transaction Execution

- Removal of an entire phase in EP, (compared to PROMPT) gives it a strong boost towards better performance. However, in sequential workload environment, (as we see in Figure 4.4) EP blocks data for a huge interval of time compared to PROMPT. So, larger number of *priority inversions* are caused by EP. As a result it performs worse than PROMPT, especially in the situation where the Resource and data contentions are very low.

  From Figure 4.4 we also see that EP cannot have much of an edge over PROMPT because, in sequential execution, the data processing takes a huge time compared to the time required for the commit processing. So the gain of EP having "one-phase less" commit-processing-overhead does not show up on the overall performance.

- In the environment, where the cohorts are run all in parallel, the total data processing time is much less than that found in corresponding sequential execution case. However the commit processing time remains unchanged. Thus commit processing has considerably greater proportional representation in influencing the overall performance. This is quite apparent in the Figure 4.5. Chiefly because of this

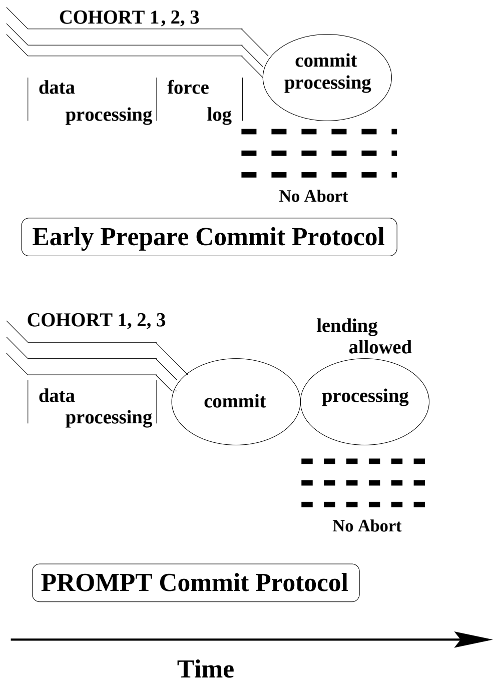

Figure 4.5: Parallel Transaction Execution

reason and the fact that EP's lock-holding period drastically reduces in parallel execution, EP claims to perform reasonably better than PROMPT. The duration of lock-holding is shown in the Figure 4.4 and Figure 4.5 by highlighting the time intervals where No Aborts are allowed. Also, in the figures, the intervals are marked where lending is allowed for a cohort.

- A very strong feature of PROMPT, namely *Active Abort*, which helps PROMPT to outperform all the other two-phase protocols, is inherently present in EP. In EP, a cohort can be aborted only when it is executing, or force-writing the `PREPARE` log record. Anyway, they will get aborted, if at all, before the stage of sending WORKDONE to the master. Thus they inform the master immediately as they get aborted. So, the *Active Abort* feature is present in EP, implicitly.

- In the situation of very high load, the number of active transactions in the system is large and consequently the system offers higher data and resource contention. PROMPT gives a bigger time-window for priority abort. Thus, it gets larger number of restarts, making more transactions to remain "active" in the system. Thus the resource and data contention also increases. In such situation *wasteful restarts* reach a peak. EP performs extremely well in this region. Its "long locking" feature stops it from having many wasted restarts. This helps EP to have better resource utilization and to face less data and resource contention.

We believe that our study and thorough quantitative analysis of the one-phase commit protocol EP completes the work started by [Gup97]. Our study considers an extensive set of environments and system configurations. This study would help the Distributed RTDBS designers in making a conscious and more effective decision while choosing the commit protocol for their systems.

4.9 Related Work on Real-Time Commit Protocols

The design of real-time commit protocols has been investigated earlier in [DLW89, SLKS92, Yoo94, Gup97, HRG99]. [DLW89] describes a *timed 2PC* where the fate of a transaction (commit or abort) is guaranteed to be known to all cohorts of the transaction by the deadline, when there are no processor, communication, or clock faults. In case of faults, however, it is not possible to provide such guarantees, and an *exception state* is allowed which indicates the violation of the deadline. Now even if the master of a transaction is able to reach the decision by the deadline but it is not possible to convey the decision to all cohorts by the deadline, the transaction is killed (the master assumes the knowledge about the message delays etc.). Thus the primary concern in the paper was to ensure that all cohorts of a transaction reach the decision before the deadline expires, even at the cost of more transactions eventually being killed. In [Gup97, HRG99], the focus was on improving the number of transactions that complete before their deadlines expire. Since this is of primary concern in the "firm deadline" application framework, (adhering to the commitment semantics of the transactions as explained in Section 4.3) we chose to follow this in our work. One more reason for following this model is that, they did not assume any guarantee from the system for the services offered (e.g., messages, data access from the disk, etc.), unlike [DLW89] where such guarantees are fundamental to the design of the protocol.

[SLKS92, Yoo94] are based on a common theme of allowing individual sites to *unilaterally* commit—the idea is that unilateral commitment results in greater timeliness of actions. If it is later found that the decision is not consistent globally, "compensation" transactions are used to rectify the errors.

While the compensation-based approach certainly appears to have the potential to improve timeliness, yet there are quite a few practical difficulties. First, the standard notion of transaction atomicity is not supported—instead, a "relaxed" notion of atomicity [KLS90, LKS91b, LKS91a] is provided. Second, the design of a compensating transaction is an application-specific task since it is based on the application semantics. Third, the compensation transactions need to be designed in advance so that they can be executed as soon as errors are detected—this means that the transaction workload must be fully characterized *a priori*. Fourth, "real actions" [Gra81] such as firing a weapon or dispensing cash may not be compensatable at all [LKS91b].

From a performance viewpoint also, there are some difficulties: First, the execution of compensation transactions is itself an additional burden on the system. Second, it is not clear as to how the database system should schedule compensation transactions relative to normal transactions. Finally, no performance studies are available to evaluate the effectiveness of this approach.

Due to the above limitations of the compensation-based approach, we have chose to follow the considerations of [HRG99].

In a sense, the study pursued in [HRG99] was incomplete. In that study, the authors expected the one-phase commit protocols to perform poorly in RTDBS and so, ignored it in its entirety. Their logic behind this was: long un-breakable locks of one phase protocols would surely result in poor real-time performance. Although their intuition was partially correct, they did not consider that the removal of an entire phase could help one-phase protocols a lot, especially in the environment where cohorts are executed in parallel. Also, they did not consider the positive effect of "long locking" of EP in case of very high workloads where priority aborts mostly result in "wasted restarts".

CHAPTER

5

The Simulation Model

To evaluate the real-time performance of the various commit protocols described in the previous chapter, we have used a detailed simulation model based on an *open* queuing model of a distributed real-time database system. The simulation model is based on a loose combination of the distributed database model described in [CL88] and the real-time database model described in [HCL92]. The model consists of a database that is distributed, in a non-replicated manner, over N sites connected by a network. Each site in the model has six components: a *source* which generates transactions; a *transaction manager* which models the execution behavior of the transaction; a *concurrency control manager* which implements the concurrency control algorithm; a *resource manager* which models the physical resources; a *recovery manager* which implements the details of commit protocols; and a *sink* which collects statistics on the completed transactions. In addition, a *network manager* models the behavior of the communications network.

In the following sections, we describe various components of the simulation model. Subsequently, we describe the execution pattern of a typical transaction. A summary of the parameters used in the model are given in Table 5.1.

Material for this chapter is taken from [Gup97].

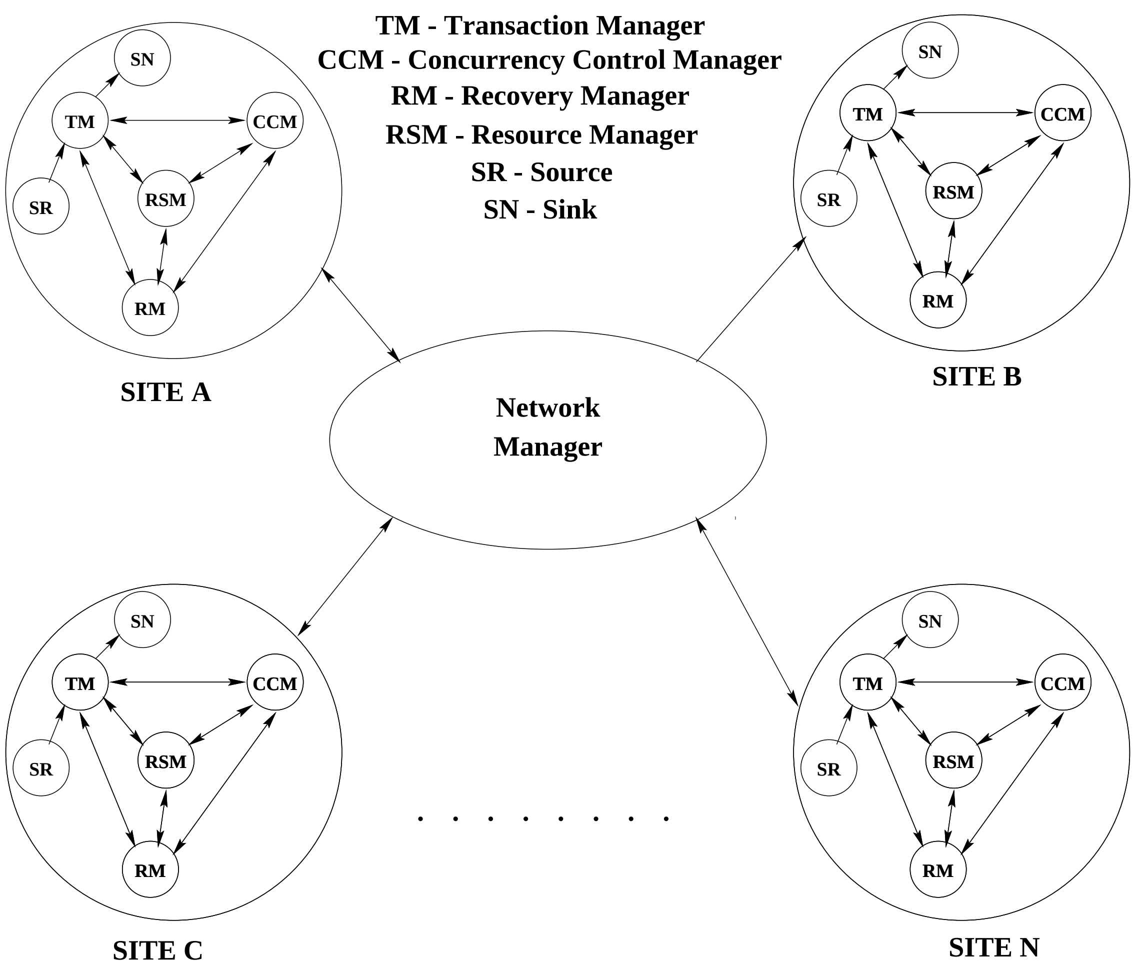

Figure 5.1: Distributed DBMS Model Structure

Table 5.1: Simulation Model Parameters

$NumSites$	Number of sites in the database
$DBSize$	Number of pages in the database
$ArrivalRate$	Transaction arrival rate / site
$SlackFactor$	Slack Factor in Deadline formula
$TransType$	Transaction Type (Sequential or Parallel)
$DistDegree$	Degree of Distribution
$CohortSize$	Average cohort size (in pages)
$UpdateProb$	Page update probability
$NumCPUs$	Number of processors per site
$NumDataDisks$	Number of data disks per site
$NumLogDisks$	Number of log disks per site
$PageCPU$	CPU page processing time
$PageDisk$	Disk page access time
$MsgCPU$	Message send / receive time
$BufHit$	Probability of finding a requested page in the buffer

5.1 Database and Workload Model

The database is modeled as a collection of $DBSize$ pages that are uniformly distributed across all the $NumSites$ sites. At each site, transactions arrive in an independent Poisson stream with rate $ArrivalRate$, and each transaction has an associated firm deadline. The deadline is assigned using the formula $D_T = A_T + SlackFactor * E_T$, where D_T, A_T and E_T are the deadline, arrival time and execution time, respectively, of transaction T, while $SlackFactor$ is a constant that provides control over the tightness/slackness of transaction deadlines. The execution time is the total service time at the resources that the transaction requires for its execution.[1] Note that although the workload generator knows E_T, the system does *not* have any knowledge about E_T.

Each transaction in the workload has the "single master, multiple cohort" structure described in Section 4.1. The number of sites at which each transaction executes is specified by the $DistDegree$ parameter. The master and one cohort reside at the site where the transaction is submitted whereas the remaining $DistDegree - 1$ cohorts are set

[1] Since the resource time is a function of the number of messages and the number of forced-writes, which differ from one commit protocol to another, we compute the execution time assuming execution in a *centralized* system.

up at different sites chosen at random from the remaining $NumSites - 1$ sites. At each of the execution sites, the number of pages accessed by the transaction's cohort varies uniformly between 0.5 and 1.5 times $CohortSize$. These pages are chosen randomly from among the database pages located at that site. A page that is read is updated with probability $UpdateProb$. A transaction that is aborted due to a data conflict is restarted immediately and makes the same data accesses as its original incarnation.

5.2 System Model

The physical resources at each site consist of $NumCPUs$ processors, $NumDataDisks$ data disks, and $NumLogDisks$ log disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemptions being based on transaction priorities. Each of the disks has its own queue and is scheduled according to a Head-Of-Line (HOL) policy, with the request queue being ordered by transaction priority. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively. A data page request can be satisfied by the buffer manager, or the page may have to be accessed from the disk. $BufHit$ parameter gives the probability of finding a page in the buffer pool.

The communication network is simply modeled as a switch that routes messages since we assume a local area network that has high bandwidth. However, the CPU overheads of message transfer are taken into account at both the sending and the receiving sites. This means that there are two classes of CPU requests—local data processing requests and message processing requests. We do not make any distinction, however, between these different types of requests and only ensure that all requests are served in priority order. The CPU overheads for message transfers are captured by the $MsgCPU$ parameter.

5.3 Priority Assignment

As mentioned earlier, transactions in a RTDB system are typically assigned priorities in order to minimize the number of missed deadlines. In our model, all the cohorts of a transaction inherit the parent transaction's priority. Further, this priority, which is assigned at arrival time, is maintained throughout the course of the transaction's existence in the system (including the commit processing stage, if any). The processing of messages sent on behalf of a transaction also occurs at the priority of the transaction. The transaction priority assignment policy used is the widely-used *Earliest Deadline* [LL73]—transactions with earlier deadlines have higher priority than transactions with later deadlines.

5.4 Concurrency Control

The 2PL High Priority scheme [AGM88] is used for concurrency control of conflicting transactions. A concurrency control request from a high priority cohort for a data item held by a low priority cohort is satisfied by aborting the low priority cohort. If a low priority cohort requests a data item held by a high priority cohort, it is made to wait until the high priority cohort releases its lock from the data item (or allows optimistic access, as in PROMPT). If a data item is locked by some cohort in the read mode, a cohort's request for sharing the lock is granted only if no cohort with a priority higher than that of the requesting cohort is waiting in the lock queue to access the data in an exclusive (write) mode. A cohort in the prepared state cannot be aborted due to concurrency reasons. On receipt of the PREPARE message from the master, a cohort releases all its read locks but retains its update locks until it receives and implements the global decision from the master. Due to the fact that 2PL-HP protocol is used to resolve the conflict between transactions, and that the transactions are assigned unique priorities at the arrival time, there is no possibility of the transactions forming a deadlock.

5.5 Execution Model

A transaction, at its arrival time, is assigned the set of sites where it has to execute and the data pages that it has to access at each of these sites. The deadline of the transaction is assigned based on the formula described in Section 5.1. The parameter $TransType$ specifies whether the transaction will execute in a sequential or parallel fashion. The master is then started up at the originating site, forks off a local cohort and sends messages to initiate each of its cohorts at the remote participating sites. Each cohort makes a series of read and update accesses. A read access involves a concurrency control request to obtain access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. We do not explicitly model concurrency control cost parameters in our model since we assume that the overheads of performing concurrency control are small compared to data processing times. Update requests are handled similarly except for their disk I/O—the writing of the data pages takes place asynchronously after the transaction has committed. We assume sufficient buffer space to allow the retention of data updates until commit time.

The commit protocol is initiated when the transaction has completed its data processing. If the transaction's deadline expires either before this point, or before the master has written the global decision log record, the transaction is killed (the precise semantics of firm deadlines in a distributed environment are defined in Section 4.3). If the master has written the commit decision log record before the expiry of the deadline, the transaction is said to be committed.

5.6 Logging

With regard to logging costs, we explicitly model only *forced* log writes since they are done synchronously and suspend transaction operation until their completion. The cost of each forced log write is the same as the cost of writing a data page to the disk. The overheads of flushing the transaction execution log records (i.e., WAL) are not modeled since we do not expect them to affect the relative performance behavior of the commit protocols discussed in our study.

CHAPTER

6

Experiments and Results

We conducted an extensive set of experiments to evaluate the performance of the one-phase real-time commit protocol EP and compare it with a representative set of protocols, briefly discussed in chapter 4. Wherein we restricted ourselves to the classic two-phase commit protocol and the optimistic protocol PROMPT [HRG99] in addition to the one-phase protocol EP. A detailed model of a firm deadline distributed RTDB system as described in chapter 3 was considered for this purpose. The simulator (taken from [Gup97]) described in Chapter 5 has been used. In this chapter, we present the results of these experiments and discuss the performance implications of supporting transaction atomicity in a real-time environment.

6.1 Performance Metric

The performance metric of our experiments is $KillPercent$, which is the percentage of input transactions that the system is *unable* to complete before their deadlines. KillPercent values in the range of 0 to 20 percent are taken to represent system performance under "normal" loads, while KillPercent values in the range of 20 percent to 100 percent represent system performance under "heavy" loads. A long-term operating region where the miss percentage is high is obviously unrealistic for a viable distributed RTDB system. Exercising the system to high miss levels (as in our experiments), however, provides valuable information on the response of the algorithms to brief periods of stress loading.

All the KillPercent graphs in this chapter show values with each experiment having been run until at least 20000 transactions were processed by the system. Only statistically significant differences are discussed here.

The simulator[Gup97] was designed to generate a host of other statistical information, including resource utilization, number of transaction restarts, number of force log writes per transaction, page conflict probability, restart ratio, etc. These secondary measures help to explain the KillPercent behavior of the commit protocols under various workloads and system conditions.

Comparative Protocols

To help isolate and understand the effects of data distribution and commit processing on KillPercent performance, we have analyzed the behavior of an additional scenario:

Centralized System :

Here, the simulated performance of an equivalent *centralized* database system [Gup97] has been studied. A multiprocessor system having the same number of CPUs and disks, and the same amount of data, as that of the distributed system. Note that all CPUs (and the disks) in the centralized system form a single pool, resulting in better utilization due to load-balancing. The data items in the centralized system are partitioned in the same fashion as in the distributed system in order to have the similar conflict probability patterns of the transactions in both the systems. In a centralized system, messages are obviously not required and commit processing only requires force-writing a *single* `COMMIT` log record. Modeling this scenario helps to isolate the overall effect of distribution on KillPercent performance.

In the following experiments, we will refer to the performance achievable under the above scenario as **CENT** .

Table 6.1: Baseline Parameter Values

NumSites	8	*NumCPUs*	2
DBSize	2400 pages	*NumDataDisks*	3
ArrivalRate	0.5 - 10 per site	*NumLogDisks*	1
SlackFactor	4.0	*PageCPU*	5 ms
TransType	Sequential	*PageDisk*	20 ms
DistDegree	3	*MsgCPU*	5 ms
CohortSize	6 pages	*BufHit*	0.1
UpdateProb	0.5	–	–

6.2 Experiment 1: Baseline Experiment (Sequential Execution)

We begin the performance evaluation of the real-time commit protocols by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. The values of the various parameters for the baseline experiment are listed in Table 6.1. These values were chosen to ensure significant levels of both resource contention (RC) and data contention (DC) in the system, thus helping to bring out the performance differences between the various commit protocols, without having to generate very high transaction arrival rates.

Our first experiment was conducted using the default settings for all model parameters (Table 6.1), resulting in significant levels of both resource contention (RC) and data contention (DC). Here, each transaction executes in a *sequential* fashion at three sites, accessing an average of six pages at each site. The transaction also updates the pages with an *update probability* of 0.5. Each site has two CPUs, three data disks and one log disk. For this environment, Figure 6.1.a and 6.1.b show the KillPercent behaviors under normal load and heavy load conditions, respectively.

In these graphs we observe that at normal load condition, the centralized performance (CENT) is expectedly the best and PROMPT performs considerably better than 2PC . We also see that EP performs considerably worse than CENT and PROMPT falls in between. Throughout the range of normal loading PROMPT fares reasonably better than

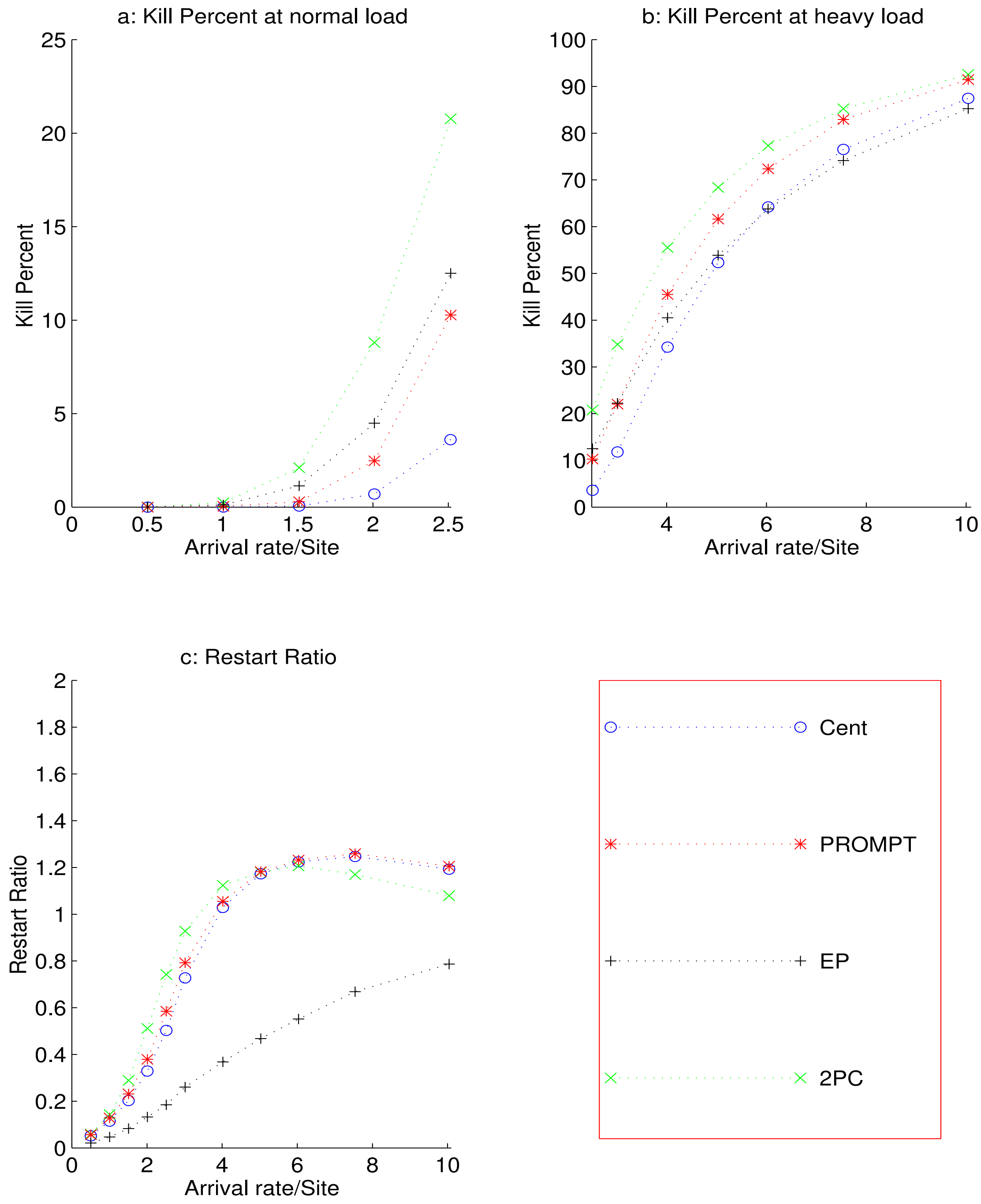

Figure 6.1: Baseline Experiment (Sequential, RC+DC)

EP.

Here we realize that optimistic borrowing is effective, especially in the low to medium loading range. At high loads, however, only a few transactions are able to make it to the commit processing phase and correspondingly there are very few lenders, leaving little opportunity for the optimistic feature to come into play.

The other important feature of PROMPT, namely active abort, helps it perform considerably better than all the conventional commit protocols like 2PC, PA and PC. But this feature is also present in EP, implicitly. Because in case of EP, priority abort is possible only when a cohort is *executing* the job requested by its master. So PROMPT does not get any edge over EP by virtue of this feature, which proved to be very strong while compared with 2PC.

In the normal load situation, the EP protocol performs badly because of the long locks it holds. In EP, once a cohort finishes its job, it goes directly to the commit phase (after force-writing a prepare log record of course) and freezes its locks. Thus it indulges in priority inversion, leading to bad performance. In contrast, PROMPT can priority-abort a cohort in voting phase, and thus helps high priority (ie. closer deadline) transactions to meet their deadlines. This goes in favor of PROMPT in addition to its Optimistic Borrowing feature.

In the heavy load situation, we observe that EP performs extremely well. It even beats the CENT, which has the best possible performance of any two-phase based protocol. This happens because, in the heavy load condition, CENT, 2PC, PROMPT keep on having too many transaction-restarts. The plots of "restart ratio", in the figure 6.1.c proves this fact. These restarts make too many active transactions to be present in the system. This effectively leads to more wasteful restarts and contributes to higher contention for data items and system resources. In this situation, the locking property of EP works in a positive sense. A considerably less number of restarts, as is seen in the figure 6.1, works towards more resource utilization and thus result in better performance. One more feature of EP implicitly helps it perform better in very high load situations. Since EP allows every successful cohort to hold its locks until its master sends decision, the transaction that has already finished larger fraction of its job, gets greater insurance towards commitment. Because, this insurance dynamically grows with the number of its successful cohorts. This phenomenon works in favor of resource utilization.

6.3 Experiment 2: Pure Data Contention (Sequential Execution)

The goal of our next experiment was to *isolate* the influence of *data contention* (DC) on the performance of the commit protocols. Studying this scenario is important because while resource contention can usually be addressed by purchasing more and/or faster resources, there do not exist any equally simple mechanisms to reduce data contention. In this sense, data contention is a more "fundamental" determinant of database system performance. Further, while abundant resources may not be typical in conventional database systems, they may be more common in RTDBS environments since many real-time systems are sized to handle transient heavy loading. This directly relates to the application-domain of RTDBSs, where functionality, rather than cost, is usually the driving consideration.

For this experiment, the physical resources (CPUs and disks) were made "infinite", that is, there is no queuing for these resources [ACL87]. The other parameter values are the same as those used in Experiment 1. The KillPercent performance results for this system configuration are presented in Figure 6.2.a and Figure 6.2.b for normal load and heavy load conditions,respectively, along with the supporting statistics "restart ratios" in Figure 6.2.c.

We observe in Figure 6.2 that the relative performance of 2PC, PROMPT and EP is qualitatively similar to that was observed in the RC+DC environment of the previous experiment. Quantitatively however, there is a considerable difference. EP performs much worse than PROMPT throughout the entire range, for both normal and heavy load situations. The gain EP had over 2PC, and PROMPT in the heavy load situation of the (RC+DC) experiment was chiefly due to its more resource utilization and facing considerably less amount of resource contention. But in this pure DC experiment, the presence of active transactions in the system, which is much more in case of PROMPT or CENT, does not contribute in any way to the resource contention. This clearly shows that so far as data contention is concerned PROMPT wins over EP with a considerable margin throughout the entire loading range.

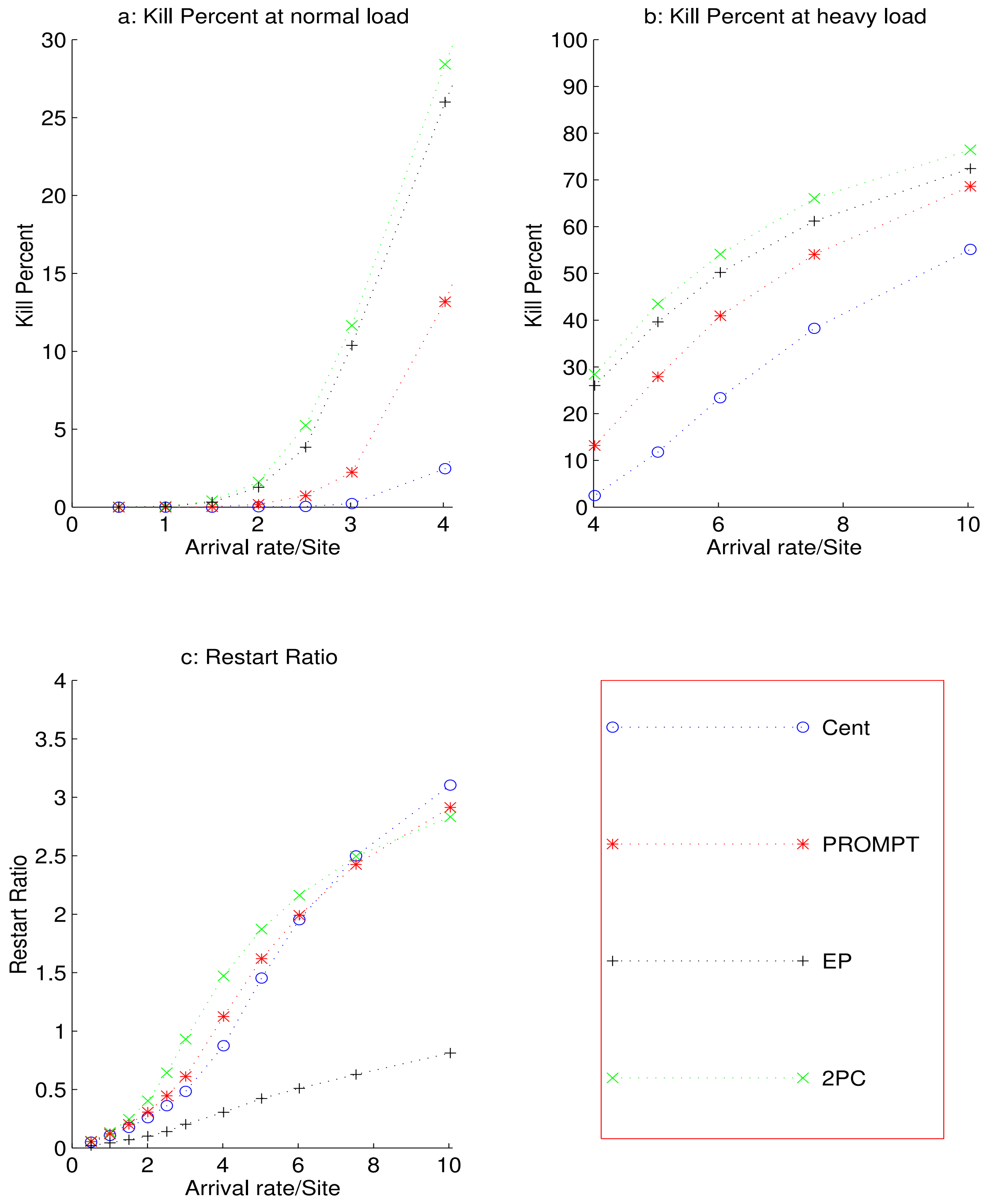

Figure 6.2: Pure Data Contention (Sequential Execution)

The above experiment indicates that when resource contention is reduced by upgrading the physical resources, it is even more important to employ protocols such as PROMPT to mitigate the ill effects of data contention and data blocking.

6.4 Experiment 3: Parallel Execution of Cohorts

In the previous experiments, the cohorts of each transaction executed in *sequence*. We also conducted similar experiments for transaction workloads with *parallel* cohort execution and we report on those results here. The performance under the RC+DC environment is shown in Figures 6.3(a,b) and we observe here that the general trends are like those seen for sequential transactions in Experiment 1. In particular, the effect of distributed commit processing on the KillPercent performance is considerably more than that of distributed data processing, as is explained in Figures 4.4 and 4.5. Here, we observe a *few changes*, as described below.

First, we observe that the differences between the performance of CENT and that of PROMPT and EP have *increased* for parallel transactions as compared to that for sequential transactions. In other words, the curves have bigger gaps in between. This is seen in the Figures 6.3(a,b). The reason for this is that the parallel execution of the cohorts reduces the transaction response time, *but the time required for the commit processing remains the same*. Therefore, the effect of the commit phase on overall transaction response time is significantly more. EP is one phase commit in contrast to the two phases of PROMPT. This helps EP gain significantly and it remains better than PROMPT throughout the entire loading range.

Second, the performance of PROMPT suffers because of its increasingly poor success in optimistic borrowings in case of parallel workload. The reason for this is, the data contention level is smaller with parallel execution than with sequential execution since locks are held for shorter times on average. Cohorts are therefore able to obtain the necessary locks sooner than in the sequential case, and hence, those that are aborted due to deadline expiry tend to make further progress than in the sequential case. This leads to a proportionally larger group of cohorts finishing their work closer to the deadline, resulting in a worse success probability due to more "unhealthy lenders".

Third, EP performed worse in Sequential workload, chiefly because of locking the data items for long and thus contributing to more numbers of priority inversion to take place. However, in the case of parallel workload, the duration of each priority inversion is

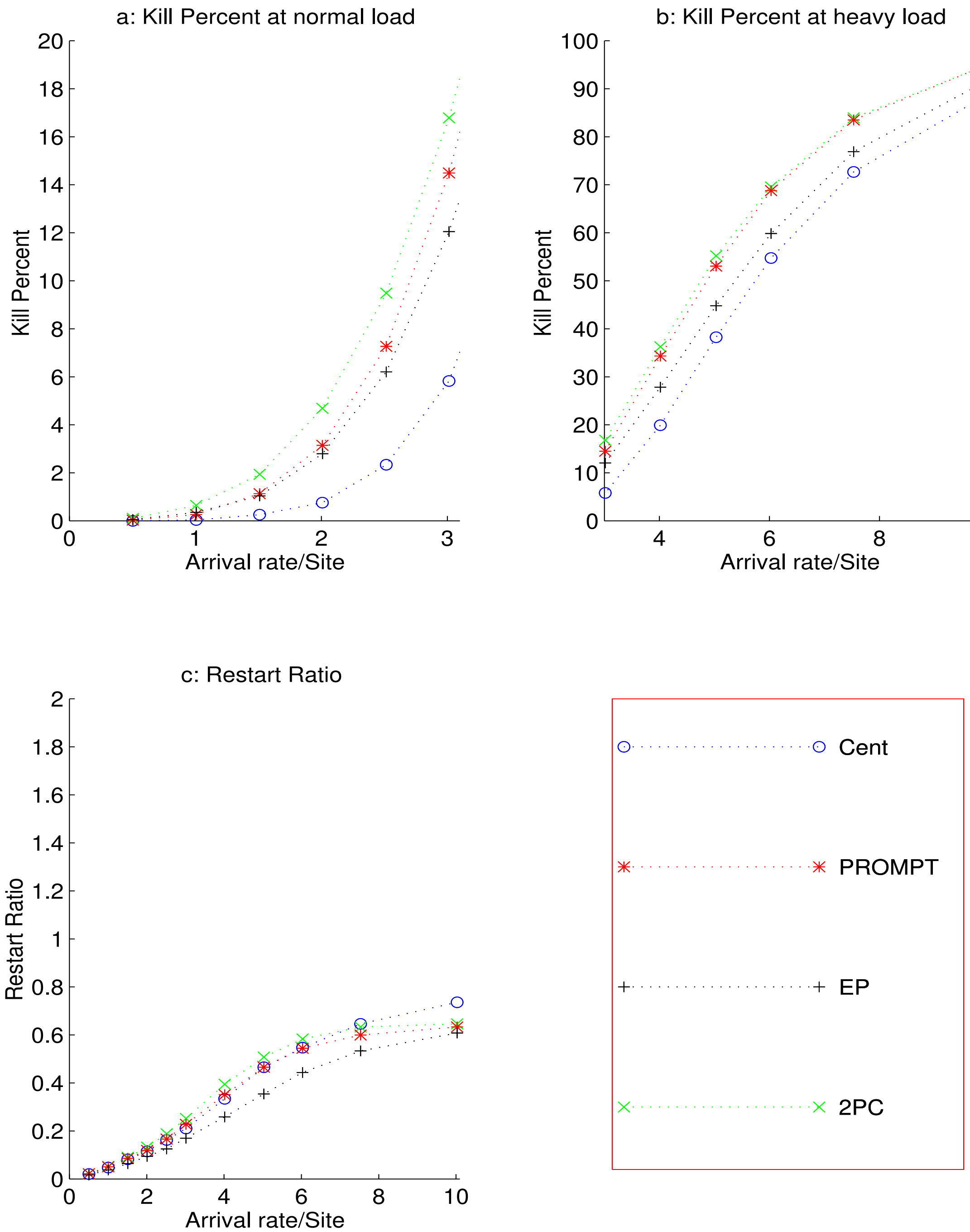

Figure 6.3: Parallel Transactions (RC + DC)

considerably less. This also makes a positive impact towards better performance of EP.

When the above experiment was conducted for the pure DC environment, we obtained Figure6.4. For 2PC and CENT the mutual difference between the curves is increased. Interestingly, the performance of PROMPT in these figures does not show the deterioration observed in the RC+DC environment. The reason is that when DC is the main performance bottleneck, PROMPT which primarily addresses DC has the maximum impact. In the absence of resource contention, many of the "unhealthy lenders" eventually commit and this gives a positive boost to PROMPT's performance. So, PROMPT gains by having more successful optimistic borrowings. This helps it to have performance considerably better than EP over a wide loading range.

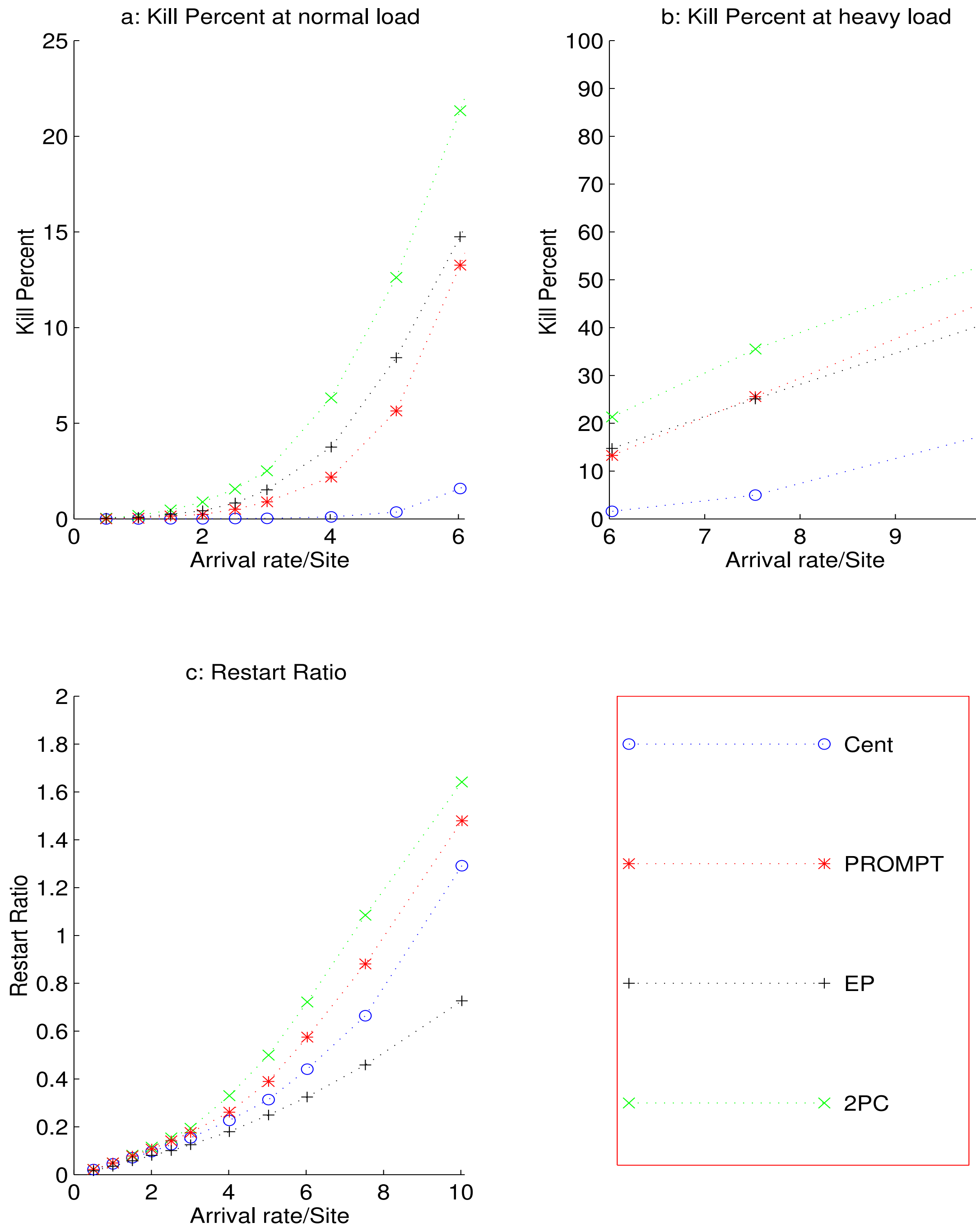

Figure 6.4: Parallel Transaction (Pure DC)

6.5 Experiment 4: Fast Network Interface (Sequential Execution)

In the previous experiments, the cost for sending and receiving messages modeled a system with a relatively slow network interface ($MsgCpu = 5\,ms$). We conducted another experiment wherein the network interface was faster by a factor of five, that is, where $MsgCpu = 1\,ms$. The results of this experiment are shown in Figure 6.5 for the RC+DC environment.[1]

The Figure 6.6 contains the results of the Pure DC experiment. In these figures, we see that the performance curves of all the protocols comes closer together (as compared to that seen in Experiments 1 and 2). This improved behavior of the protocols is only to be expected since, low message costs effectively eliminate the effect of a significant fraction of the message passing overheads involved in each protocol. Their relative performance, however, remains the same. Note also that the PROMPT protocol now provides a performance that is closer to that of CENT . We also notice that here PROMPT fares better than EP for a longer range of loading compared to our Experiment 1. Since, PROMPT involves more number of messages because of its two-phase nature, it gains more out of a faster network than EP.

[1] The default parameter settings for the RC+DC and pure DC scenarios in this experiment, as well as in the following experiments, are the same as those used in Experiment 1 and Experiment 2, respectively.

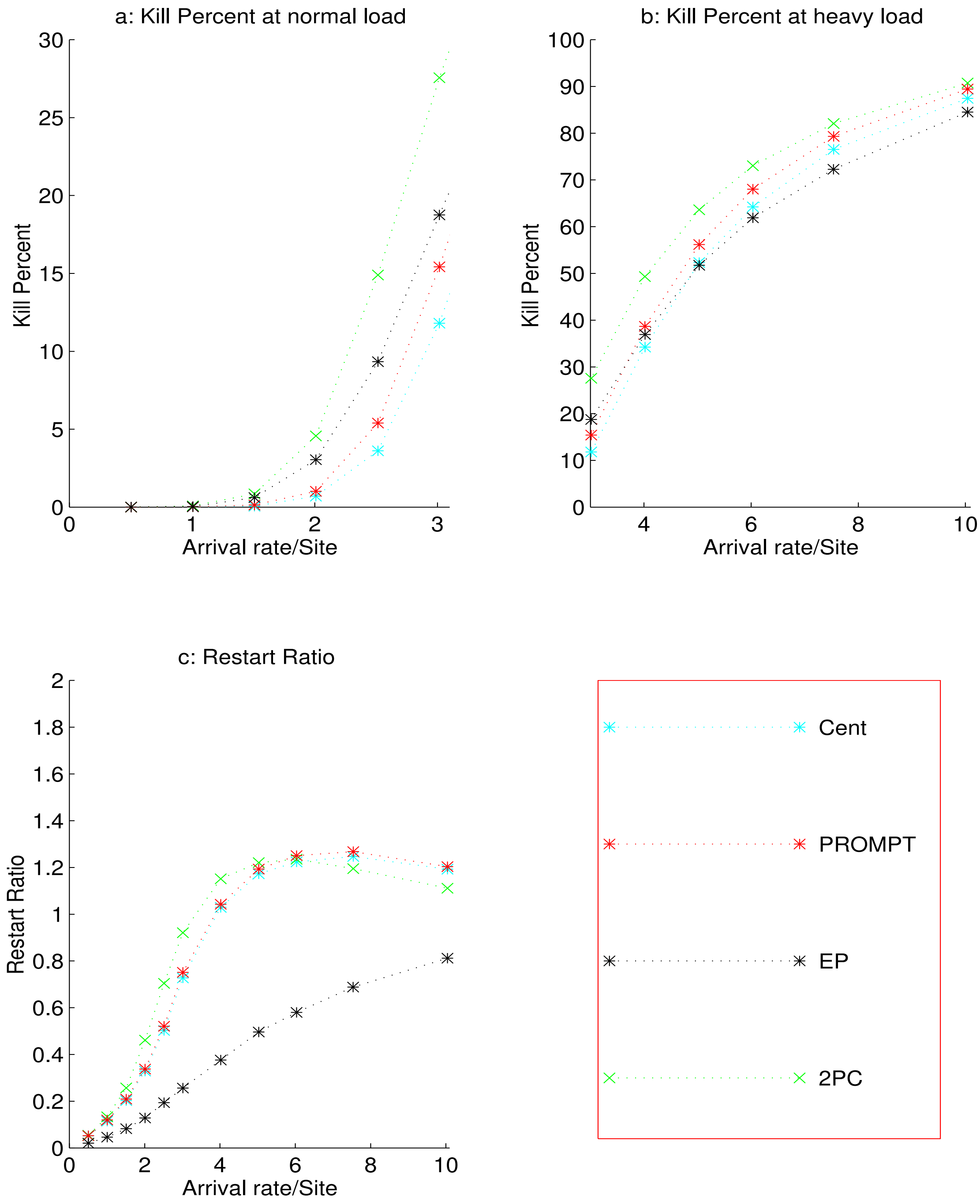

Figure 6.5: Fast Network Interface (RC+DC)

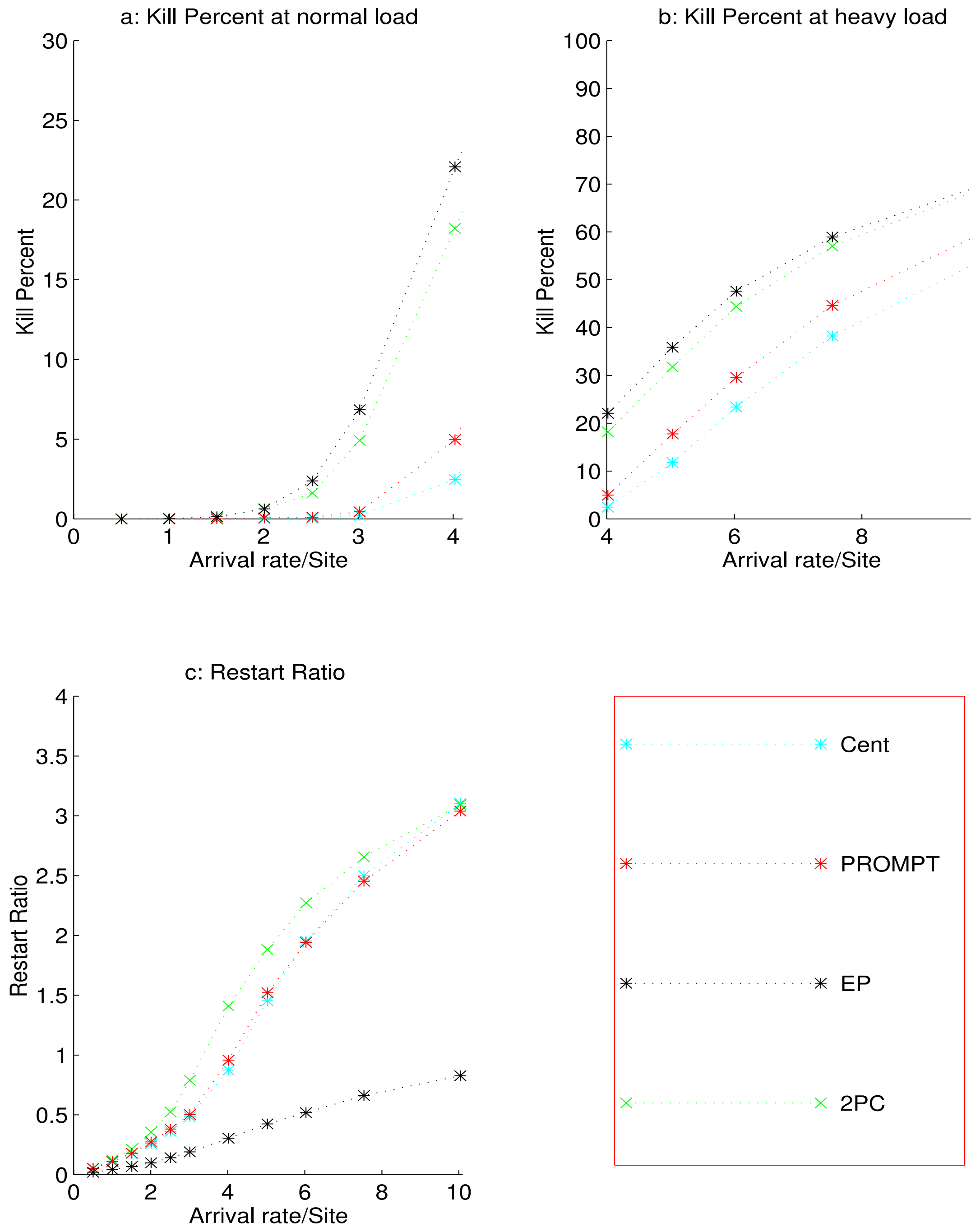

Figure 6.6: Fast Network Interface (Pure DC)

6.6 Experiment 5: Higher Degree of Distribution (Sequential Execution)

In the experiments described so far, each transaction executed on *three* sites. To investigate the impact of having a higher degree of distribution, we performed an experiments wherein each transaction executed on *six* sites. The $CohortSize$ in this experiment was reduced from 6 pages to 3 pages in order to keep the average transaction length equal to that of the previous experiments. In addition, a higher value of $SlackFactor = 6$ was used to cater to the increase in response times caused by the increased distribution.

The results of this experiment with sequential workload are shown in Figure 6.7 for the RC+DC environments. In this figure, we observe that EP performs worse than what it did in Experiment 1. Higher distribution reduces the data processing time for each individual cohort. So, the opportunity of priority abort gets reduced, this leads to greater priority inversion and worse performance. This enhanced ill effect of blocking is seen in the graphs of RC+DC environment.

This effect make EP perform worse than PROMPT, for a bigger interval of loading range in the RC+DC environment.

In the pure DC case, the effect is more prominent. Here, even in the extremely high load situation, EP continues to perform much worse throughout the entire loading range. This is expected, as the effect of "comparatively less resource contention in EP in high load range" is absent. Thus it can not mask the ill effect of blocking.

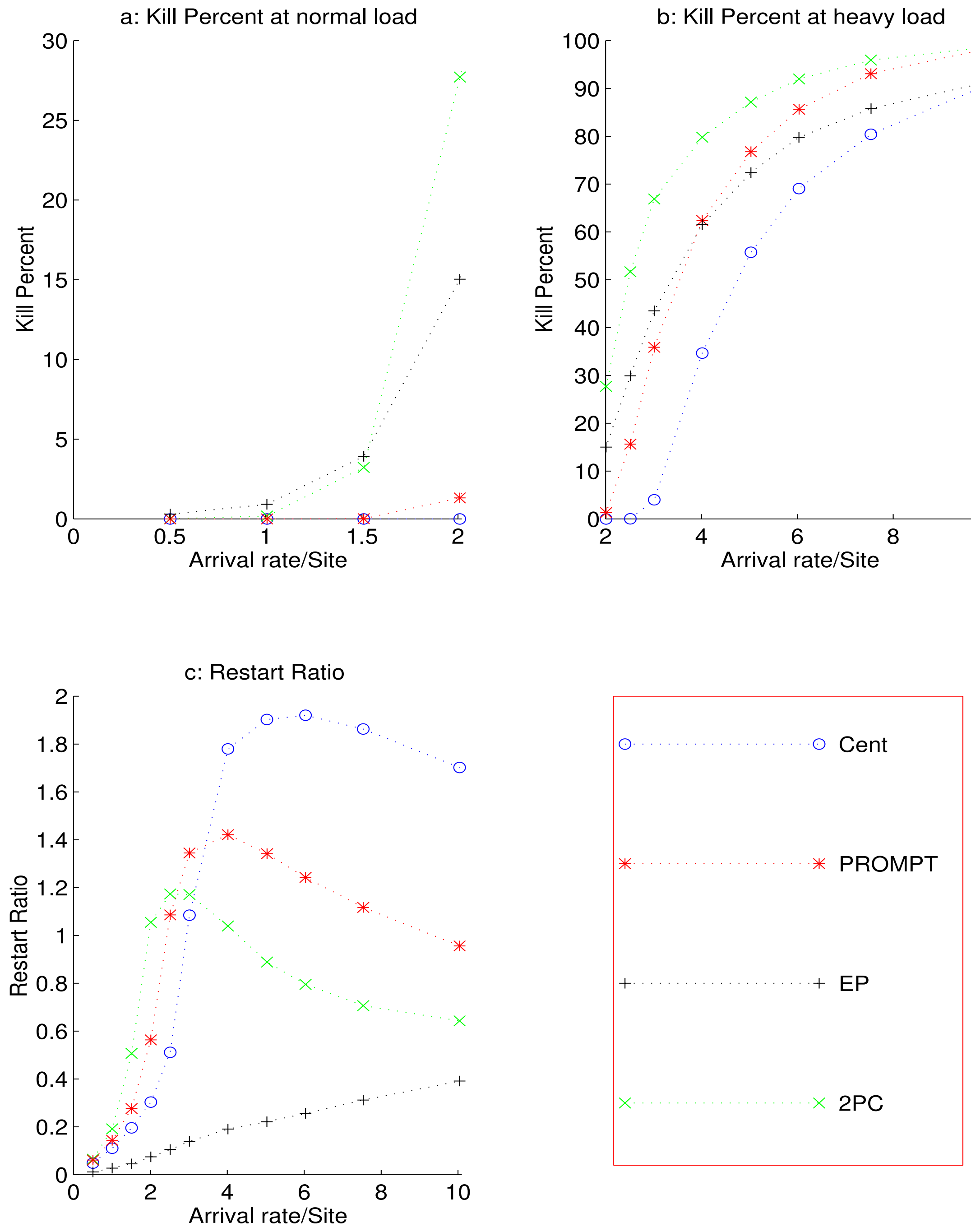

Figure 6.7: Higher Degree of Distribution (RC+DC)

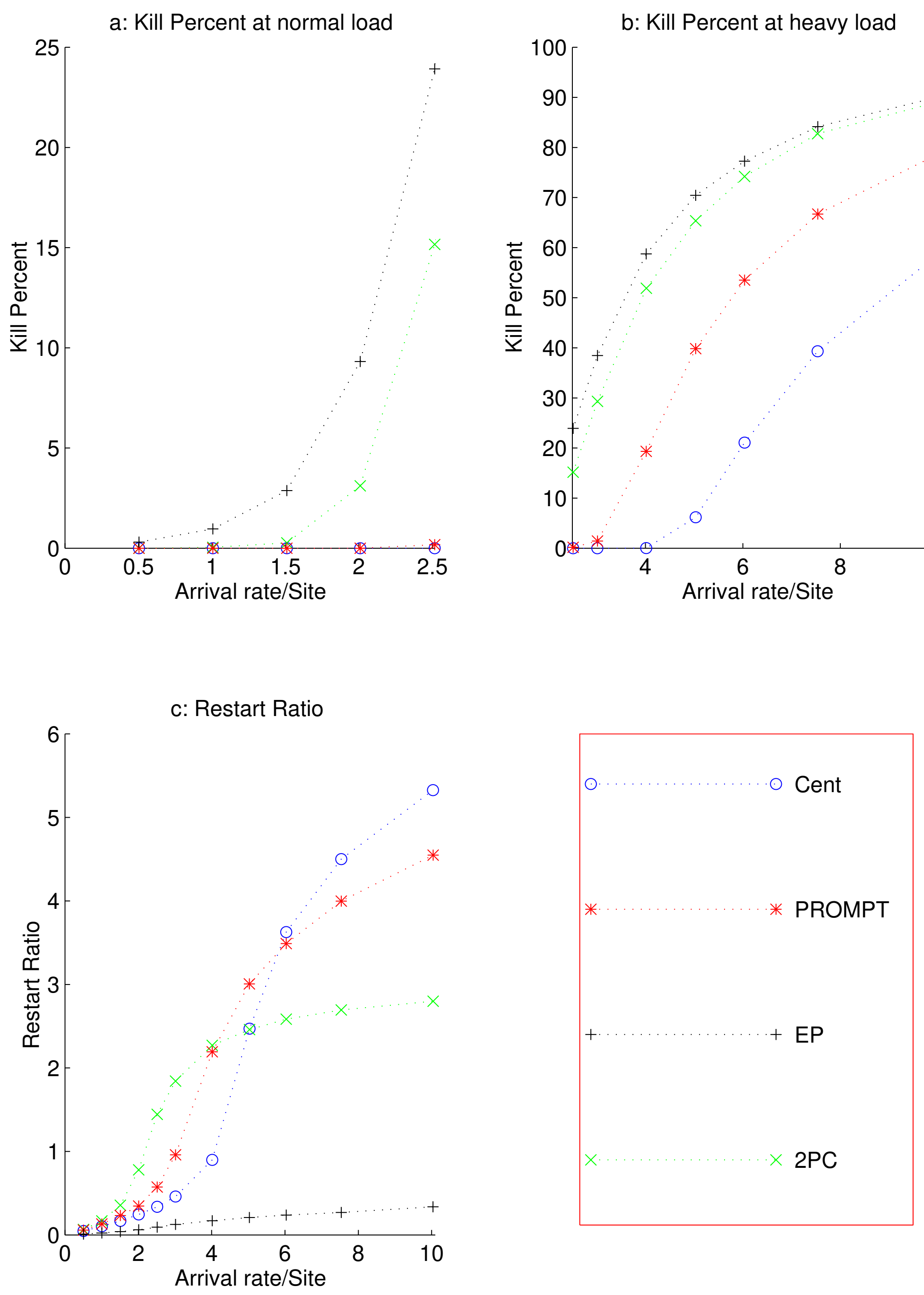

Figure 6.8: Higher Degree of Distribution (Pure DC)

6.7 Experiment 6: Higher Degree of Distribution (Parallel Execution)

The results of this experiment with parallel workload are shown in the Figure 6.9 for the RC+DC environment.

The observations are qualitatively similar to that obtained in parallel workload case in the normal distribution scenario. The higher degree of distribution, gives more parallelism to the transactions .

Figure 6.10 shows the result of this experiment with parallel workload in the Pure DC environment.

Here we see that the performance of EP is better than PROMPT. Because of higher parallelism the one-phase-commit property of EP gains much over two-phase based PROMPT especially in the very high load situation. We observed the effect of parallelism in Figures 4.4 and 4.5. The same shows enhancement in this experiment. This enhanced effect masks the effect by which PROMPT gained in the similar situation in the normal distribution scenario.

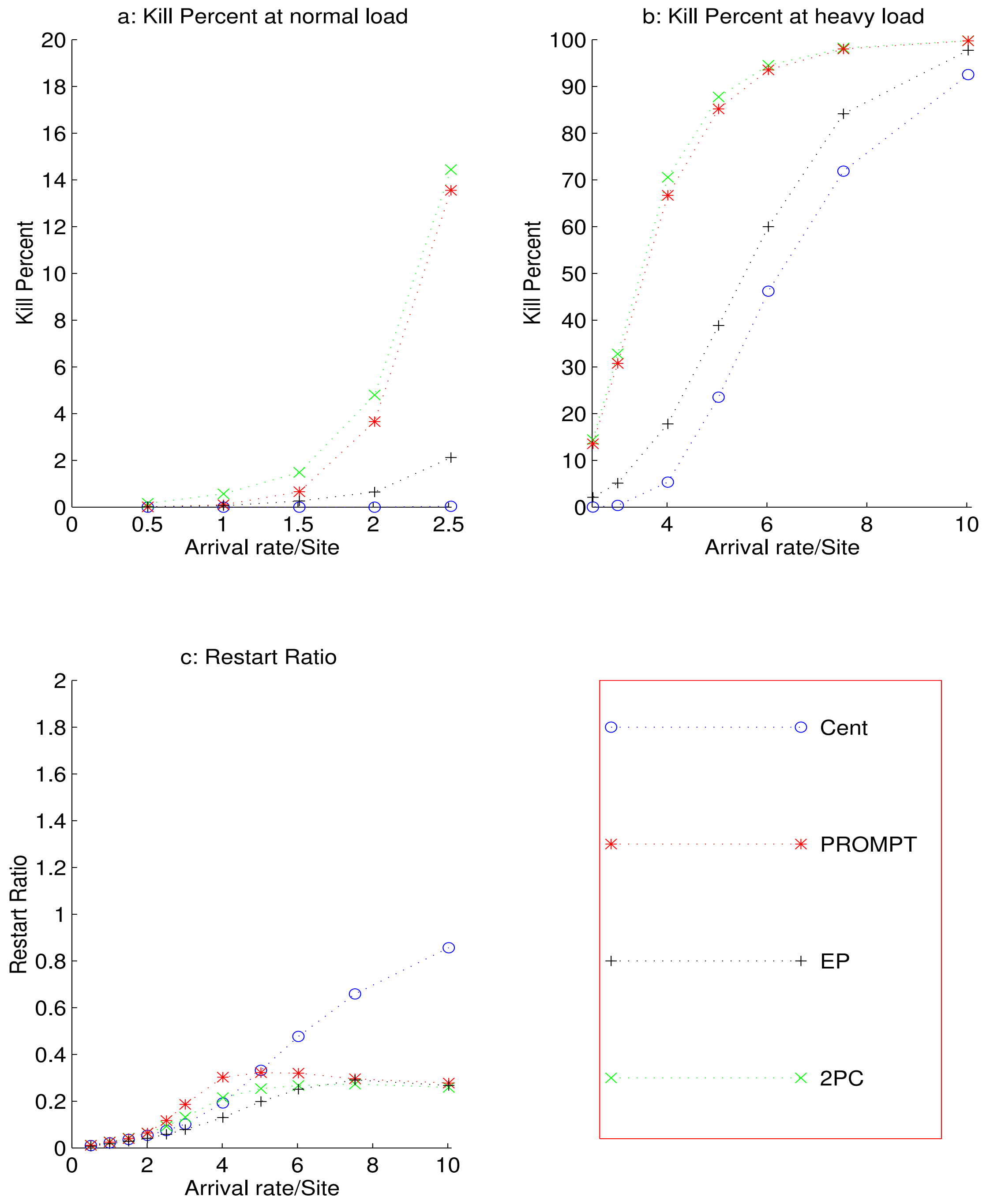

Figure 6.9: Higher Degree of Distribution (Parallel Execution: RC+DC)

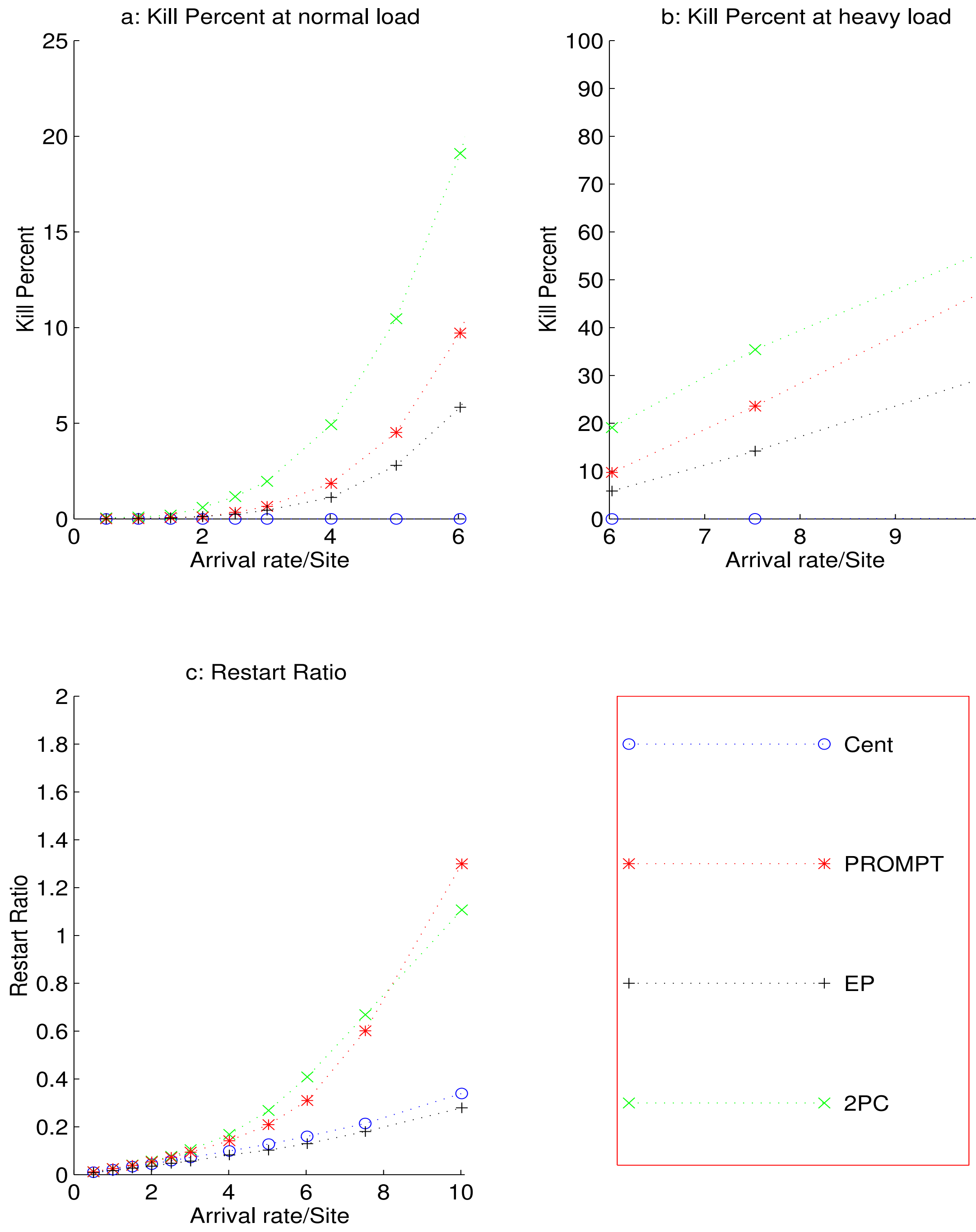

Figure 6.10: Higher Degree of Distribution (Parallel Execution: Pure DC)

CHAPTER

7

Conclusions

Although a considerable amount of research work has been done in centralized real-time database systems, comparatively little endeavor have so far been made in the field of distributed RTDB. In particular, the problem of commit processing in a distributed environment was almost neglected with the solitary exception of [Gup97]. To the best of our knowledge, there has been no quantitative study of one-phase commit protocols in distributed RTDB systems. This is what we have addressed in this thesis.

Using a detailed simulation model of a firm-deadline DRTDBS, we have made a thorough study of the one-phase commit protocol EP and analyzed its performance in a variety of environments and system configurations. In addition, we have evaluated the performance of the classic two-phase commit (2PC) protocol and the optimistic protocol PROMPT with respect to the number of killed transactions. The comparison of EP was made mostly with PROMPT, because, PROMPT is proved to be the best performing two-phase protocol in DRTDBS.

Our experiments, covering a variety of transaction workloads and system configurations, demonstrated the followings.

1. The one-phase *Early Prepare* protocol significantly reduces the overhead of commit processing by eliminating an entire phase, making it especially attractive in the real-time environment.

2. EP implicitly holds one strong feature of PROMPT, namely active abort. As a

result, EP outperforms PROMPT - the best-performing two-phase protocol when the cohorts of a distributed transaction execute in parallel.

3. However, the performance of EP is rather poor in the environment where the cohorts of a distributed transaction execute sequentially. Here the effect of prolonged unbreakable data-locks, causing greater priority inversions, becomes dominant and this makes EP perform poorly compared to the other protocols.

4. In case of *very high* workload, in the environment of sequential execution, EP performs extremely well compared to any two-phase protocol in the presence of both resource contention and data contention. In fact, here EP performs even better than a centralized system. Here the "long locking" feature of EP works positively by preventing "wasted restarts" and thereby reducing "resource wastage".

5. In the very high workload situation, as the number of restarts is low, the active transactions present in the system is much less in case of EP than in case of any two-phase protocol allowing huge amount of restarts. By virtue of this fact, EP faces considerably less resource contention.

   Therefore, in very high load situation, EP is always preferred over any two-phase protocol.

7.1 Future Work

It will be interesting to actually implement **EP** and **PROMPT** in a commercial system where EP will be used in *parallel workload environment* and PROMPT will be used in *sequential workload environment*. For extremely high workload cases, EP should be chosen always.

This work can be extended to address the security considerations in Multi-Level Secure(MLS) distributed RTDBS [RBJM96, RMJB98]. In this direction some algorithms have already been proposed in conventional distributed database systems, like : Secure Early Prepare (SEP)[ABJ94], and Advanced Secure Early Prepare (ASEP)[RMJB98].

However there is no quantitative study of their performances in the MLS distributed RTDB systems domain. So a quantitative study of them is of great future importance.

As a new proposal, a combination of PROMPT and EP (namely P-EP) seems to be quite promising.

References

[ABJ94] V. Atluri, E. Bertino, and S. Jajodia. Degree of Isolation, Concurrency Control Protocols and Commit Protocols. In *Database Security, VIII: Status and Prospects*, pages 259 – 273. North Holland, Amsterdam, 1994.

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4), 1987.

[AGM88] Robert K. Abbott and Hector Garcia-Molina. Scheduling Real-Time Transactions: a Performance Evaluation. In *Proceedings of the 14th International Conference on Very Large Data Bases*, August 1988.

[AGM92] Robert K. Abbott and Hector Garcia-Molina. Scheduling Real-Time Transactions: a Performance Evaluation. *ACM Transactions on Database Systems*, 17(3), 1992.

[Bes96] Azer Bestavros. Advances in Real-Time Database Systems Research. *Special Section on Advances in Real-Time Database Systems, ACM SIGMOD RECORD*, 25(1), 1996.

[BH95] Mikael Berndtsson and Jörgen Hansson, editors. *Active and Real-Time Database Systems (ARTDB-95), Proceedings of the First International Workshop on Active Real-Time Database Systems, Skövde, Sweden*. Workshops in Computing. Springer-Verlag, London, June 1995.

[BH96] Mikael Berndtsson and Jörgen Hansson. Workshop Report: The First International Workshop on Active and Real-Time Database Systems (ARTDB-95). *ACM SIGMOD RECORD*, 25(1), 1996.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[CL88] Michael J. Carey and Miron Livny. Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication. In *Proceedings of the 14th International Conference on Very Large Data Bases*, August 1988.

[DLW89] Susan Davidson, Insup Lee, and Victor Wolfe. A Protocol for Timed Atomic Commitment. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989.

[GHR97] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting Commit Processing Performance in Distributed Database Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, May 1997.

[Gra78] Jim Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*. 60, Springer-Verlag, Berlin, 1978.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, September 1981.

[Gup97] Ramesh Kumar Gupta. Commit Processing in Distributed On-Line and Real-Time Transaction Processing Systems. Master's thesis, Supercomputing Education and Research Centre, Indian Institute of Science, Bangalore, India, 1997.

[HCL90] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, December 1990.

[HCL92] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Data Access Scheduling in Firm Real-Time Database Systems. *The Journal of Real-Time Systems*, 4(3), 1992.

[HRG98] J. Haritsa, K. Ramamritham, and R. Gupta. Characterisation and Optimization of Commit Processing Performance in Distributed Database Systems. Technical report, University of Massachussets, March 1998.

[HRG99] J. Haritsa, K. Ramamritham, and R. Gupta. The PROMPT Real-Time Commit Protocol. In *IEEE Transactions on Parallel and Distributed Systems*, 1999.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, August 1990.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 1978.

[LKS91a] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. A Theory of Relaxed Atomicity. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, August 1991.

[LKS91b] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, May 1991.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1), 1973.

[LS76] B. Lampson and H. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Xerox, Palo Alto Research Center, California, 1976.

[ÖV91] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[RBJM96] I. Ray, E. Bertino, S. Jajodia, and L. V. Mancini. An advanced commit protocol for MLS distributed database systems. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security, New Delhi,India*, 1996.

[RMJB98] I. Ray, L. V. Mancini, S. Jajodia, and E. Bertino. ASEP: A Secure and Flexible Commit Protocol for MLS Distributed Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 1998.

[SBCM95] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment . *Journal of Distributed and Parallel Databases. Also in IEEE-ICDE,april 1993*, 3(4), 1995.

[SC90] James W. Stamos and Flaviu Cristian. A Low-Cost Atomic Commit Protocol. In *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems*, October 1990.

[SC93] James W. Stamos and Flaviu Cristian. Coordinator Log Transaction Execution Protocol. *International Journal on Distributed and Parallel Databases*, 1(4), 1993.

[SJR91] Peter M. Spiro, Ashok M. Joshi, and T. K. Rengarajan. Designing an Optimized Transaction Commit Protocol. *Digital Technical Journal*, 3(1), 1991.

[SK92] Sang H. Son and Spiros Kouloumbis. Replication Control for Distributed Real-Time Database Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

[SLKS92] Nandit Soparkar, Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. Adaptive Commitment for Real-Time Distributed Transactions. Technical Report TR-92-15, Department of Computer Science, University of Texas-Austin, 1992.

[Son88] Sang H. Son, editor. *ACM SIGMOD RECORD, Special Issue on Real-Time Database Systems*, 17(1), 1988.

[Son90] Sang H. Son. Real-Time Database Systems: A New Challenge. *Data Engineering*, 13(4), 1990.

[SRL87] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report CMU-CS-87-181, Carnegie Mellon University, 1987.

[SRL88] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Concurrency Control for Distributed Real-Time Databases. *ACM SIGMOD RECORD*, 17(1), 1988.

[UB92] Özgür Ulusoy and Geneva G. Belford. Real-Time Lock-Based Concurrency Control in Distributed Database Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

[Ulu92] Özgür Ulusoy. Current Research on Real-Time Databases. *ACM SIGMOD RECORD*, 21(4), 1992.

[Ulu94a] Özgür Ulusoy. A Study of Two Transaction Processing Architectures for Distributed Real-Time Database Systems. Technical Report BU-CEIS-94-22, Department of Computer Engineering and Information Science, Bilkent University, 1994.

[Ulu94b] Özgür Ulusoy. Research Issues in Real-Time Database Systems. Technical Report BU-CEIS-94-32, Department of Computer Engineering and Information Science, Bilkent University, 1994.

[Yoo94] Yong Ik Yoon. *Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems*. PhD thesis, Department of Information and

Communication Engineering, Korea Advanced Institute of Science and Technology, 1994.